Productividad con

Y

Jose Ignacio Paris Prieto
https://www.linkedin.com/in/jiparis
@jiparis

Capítulo 2

- *¿Qué aporta angular?*
- *Bindings*
- *Módulos y controladores*
- *Filtros*
- *Servicios*
- *Navegación y rutas*
- *Templates*
- *REST y promesas*
- *UI-Bootstrap*
- *Directivas*
- *Custom directives*

Aplicaciones JS empresariales ¿Qué necesito?

- Quiero una arquitectura en JavaScript
- Tener separación de capas y desacoplamiento
- Quiero un sistema modular, y usar IOC
- Quiero hacer testing
- Quiero realizar una integración contínua de mis desarrollos

¿Qué aporta AngularJS?

- Single Page Application

  Una sóla página. El resto de datos y contenido bajo demanda
  Mayor fluidez y usabilidad. No hay flickering

¿Qué aporta AngularJS?

- Single Page Application
- Arquitectura JavaScript

> En AngularJS existen **servicios, factorías, controladores, vistas**
> Implementa patrones como la IOC.

¿Qué aporta AngularJS?

- Single Page Application
- Arquitectura JavaScript
- Gestión de datos y conexión a servicios

1 y 2 way bindings.
Gestión de formularios y validación

¿Qué aporta AngularJS?

- Single Page Application
- Arquitectura JavaScript
- Gestión de datos y conexión a servicios
- Pensado para proyectos empresariales

Carga dinámica de módulos.
Servicios de infraestructura: conexión a servicios, caché, historia ...
Tests unitarios con Jasmine. Mocks para pruebas end2end

¿Qué aporta AngularJS?

- Single Page Application
- Arquitectura JavaScript
- Gestión de datos y conexión a servicios
- Pensado para proyectos empresariales
- Junto a TypeScript: Alta productividad

TypeScript: velocidad y control temprano de errores
AngularJS: arquitectura. Interoperabilidad con JS.

Empezamos

#00

Instalación de angular con Bower (gestor de dependencias web)

```
npm install bower -g
.bowerrc <<< {"directory": "lib"}
bower init
bower install bootstrap --save
bower install angularjs --save
```

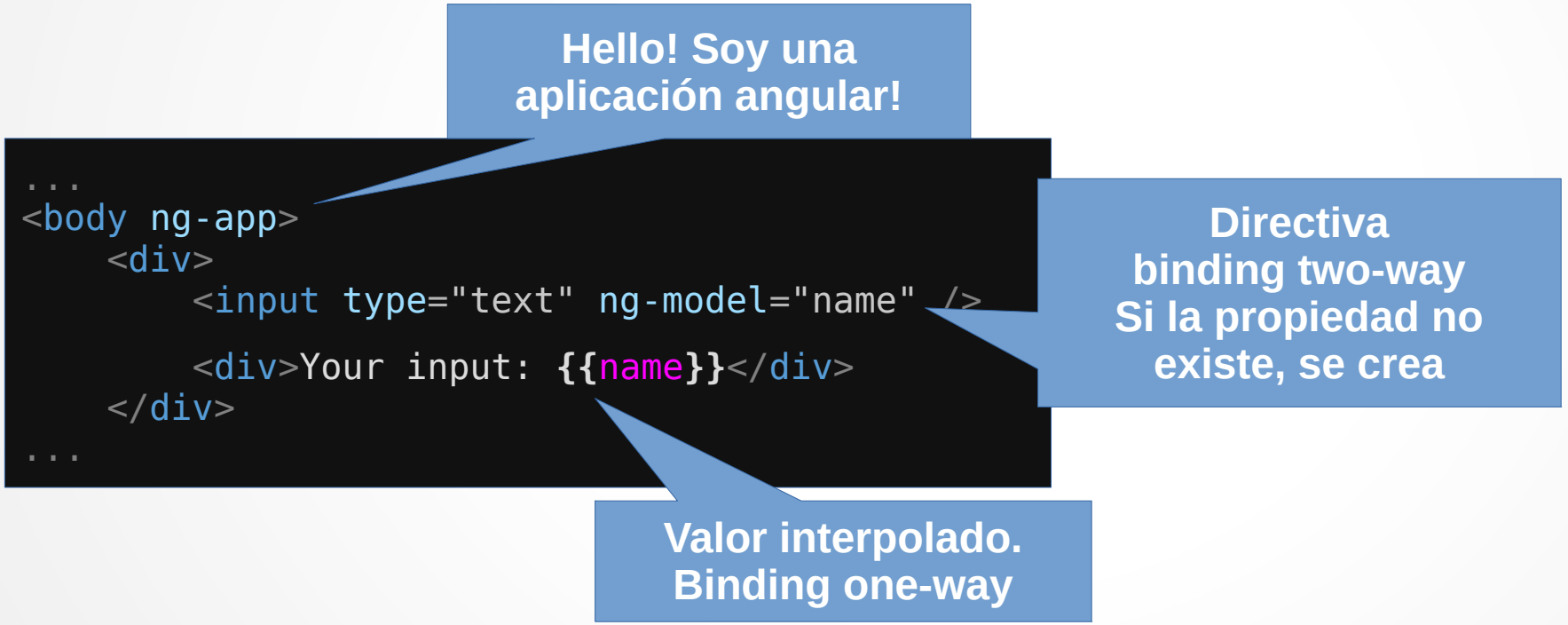
bower (idéntico a npm)
gestiona las dependencias
del proyecto
(y sus dependencias)

... más tarde: `bower install`

Binding básico

#01

- ng-app y ng-model

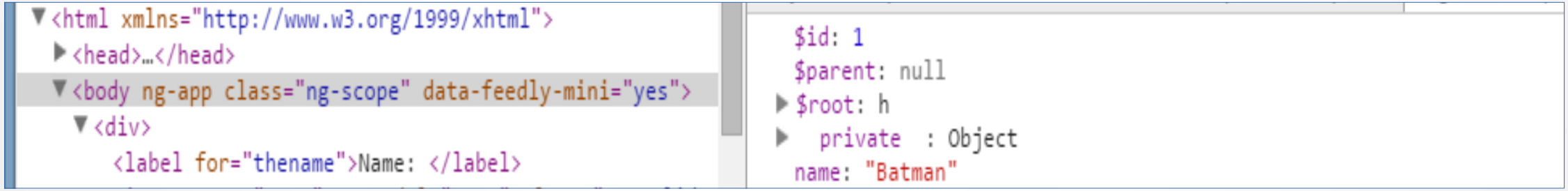


Binding básico ¿cómo funciona?

#01

Extensión de Chrome: Batarang **http://bitly.com/V8a5f1**

Toda la información relativa al binding está en el $scope



Binding básico (2)

#02

- Ng-repeat y ng-click

```
<input name="thename" type="text" ng-model="name" />
<input type="button" ng-click="people.push(name)" value="Add">

<ul ng-init="people=['Jose', 'Ana', 'Rafael', 'Superman', 'Alejandra'
    <li ng-repeat="name in people track by $index">{{name}}</li>
</ul>
```

ng-click para interacciones

ng-repeat para recorrer listas

Módulos y controladores

#03

Módulos:

- Nuestra aplicación será un módulo
- Podrá depender de otros módulos
- En general, conviene dividir la aplicación en base a módulos independientes

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

Nombre del módulo (para referenciarlo posteriormente)

Dependencias

Módulos y controladores

#03

Controllers:

- Permiten inicializar y añadir comportamiento al $scope
- Al añadir un controller, se crea un $scope hijo
- Podrá depender de otros objetos angular (generalmente servicios)

```
myApp.controller("namesController", NamesController);
```

Añade un controller al módulo

Implementación:
Función constructora en JavaScript
Clase en TypeScript

Módulos y controladores

#03

Controllers:

```
myApp.controller("namesController", NamesController);
```

```
export class NamesController {
    people = ['Jose', 'Ana', 'Rafael', 'Superman', 'Alejandra
    showImg = false;

    constructor(private $scope) {
        // setup scope

        $scope.people = this.people;
        // use arrow syntax for event handlers !!!
        $scope.addName = (name: string) => {
            this.people.push(name);
            $scope.showImg = angular.equals(name, "Batman");
        }
    }
}
```

Inyección de dependencias

Interactúa con el $scope

Módulos y controladores

#03

- **ng-controller**

Finalmente, en el HTML:

Una nota sobre la inyección de dependencias

y los ofuscadores de código

#03

```
export class NamesController {
    ...
    constructor(private $scope) {
        ...
    }
}
```

Ofuscación
(con **uglify**, por ejemplo)

```
function f9(a3){...
```

Código .js ofuscado

¿¿¿ Qué es "a3" ???

Una nota sobre la inyección de dependencias

y los ofuscadores de código

#03

Solución: *ayudar al inyector de dependencias al declarar el componente*

```
myApp.controller("namesController", ["$scope", NamesController]);
```

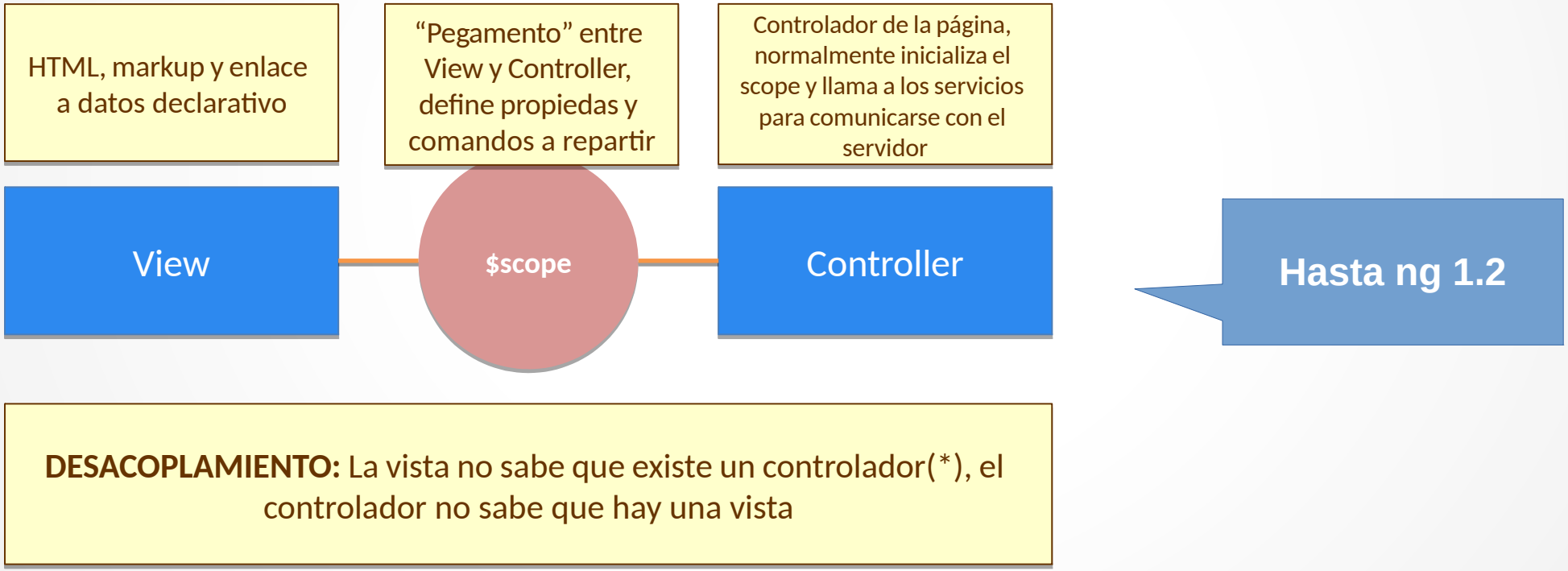
o mejor ...

```
export class NamesController {
    static $inject = [
        "$scope"
    ];
    ...
```

Todo queda encapsulado en cada componente

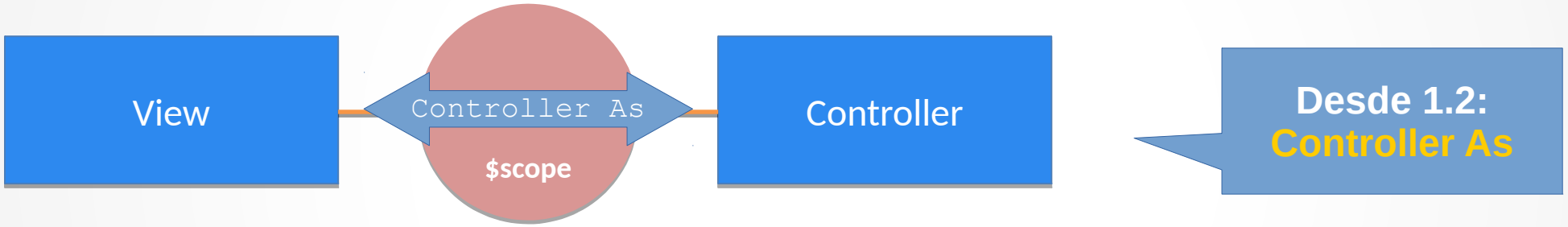
Módulos y controladores

- **$scope vs controllerAs**



Módulos y controladores

- $scope vs controllerAs



Acoplamiento: El propio controlador se inyecta en el scope, por lo que es accesible desde la vista

Módulos y controladores

- **controllerAs**


```
<div ng-controller="namesController as cont">
...
    <ul>
        <li ng-repeat="name in cont.people track by $index">{{name}}</li>
    </ul>
...
```

Herencia de **$scope**s:

```
<div ng-controller="ParentController">
    <label for="parent">Parent: </label>
    <input type="text" id="parent" ng-model="nombre" />

    <div ng-controller="ChildController">
        <label for="child1">Child 1: </label>
        <input type="text" id="child1" ng-model="nombre" />

    </div>

    <div ng-controller="ChildController">
        <label for="child2">Child 2: </label>
        <input type="text" id="child2" ng-model="nombre" />

    </div>
```

Los $scopes tienen un modelo de herencia por prototipado JavaScript.

Cualquier primitiva u objeto no existente en un $scope, se buscará en su árbol de herencia.

Filtros

Transforman los datos antes de mostrarlos

#04

```
<li ng-repeat="person in people | filter:nameFilter | orderBy:'name' track by $index">{{person.name}}</li>
```

Filtros disponibles:

- filter
- currency
- number
- date
- json
- lowercase
- uppercase
- limitTo
- orderBy

Y filtros Cusom

Filtros

Custom filters

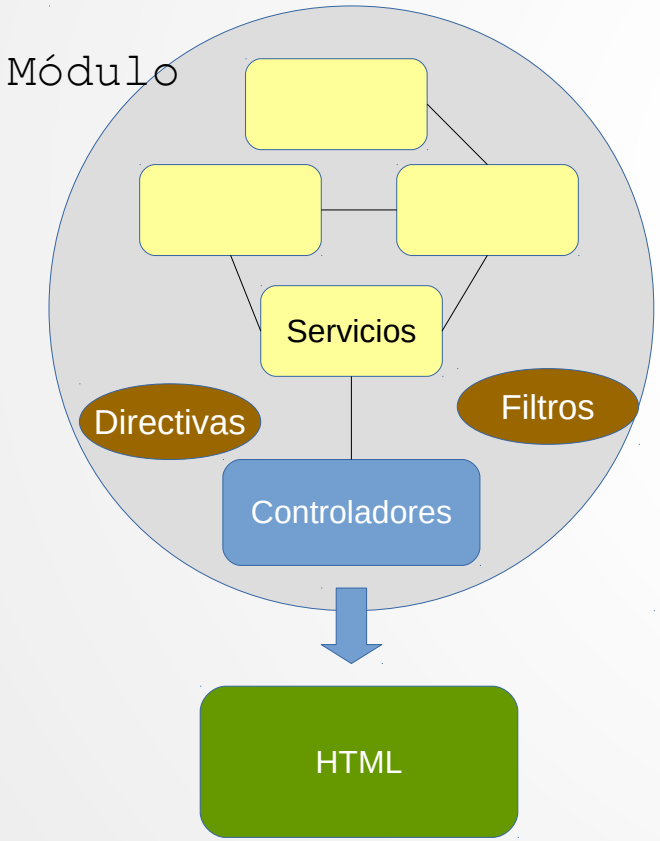
#05

```
export function DashFilter() {
    // replace all ' ' by '-'
    return (input: string) => {
        return input.replace(" ", "-");
    };
}
```

Una función factoría que devuelve la función de filtro

Servicios, factorías, proveedores ...

Montando nuestra arquitectura



Hasta ahora hemos visto:

- **Módulos**: son los contenedores de toda la aplicación
- **Controladores**. Objetos especializados para interactuar con el HTML
- **Filtros**. Modifican un objeto antes de presentarlo
- Algunas **directivas** core (crearemos directivas más adelante)

https://docs.angularjs.org/guide/providers

Servicios, factorías, proveedores ...

Qué son los servicios

- **Son singletons**
- Se instancian de forma **lazy** (sólo cuando son necesarios)
- 5 formas o recetas para crearlas:
  - **Value**
  - **Factory**
  - **Service**
  - **Provider**
  - **Constant**

https://docs.angularjs.org/guide/providers

Servicios, factorías, proveedores ...

Values

#06

```
export var maxElements = 10;
```

```
return angular.module("myApp", [])
    ...
    // value service
    .value("maxElements", maxElements)
    ...
```

```
export class NamesController {
    static $inject = [
        "$scope",
        "maxElements"
    ];
```

Útiles para valores constantes de configuración

https://docs.angularjs.org/guide/providers

Servicios, factorías, proveedores ...

Factory

#06

```
// factory service
export function RandomHeroFactory (superHeroes): any{
    return superHeroes[Math.floor(Math.random() * 5)];
}
```

```
return angular.module("myApp", [])
    ...
    // factory service
    .factory("randomHero", RandomHeroFactory)
    ...
```

```
export class SuperHeroesService {
    static $inject = [
        "superHeroes",
        "randomHero"
    ];
```

Permiten dependencias

https://docs.angularjs.org/guide/providers

Servicios, factorías, proveedores ...

Services

#06

```
export class SuperHeroesService {
    static $inject = [
        "superHeroes",
        "randomHero"
    ];

    // dependencies are inyected automatically
    constructor(private superHeroes: any[], private randomHero: any)
    }

    getHeroes(): string[] {
        return this.superHeroes;
    }

    ...
}
```

Mayor orientación a objetos (se crean con new)

```
myApp.service("heroesService", SuperHeroesService);
```

https://docs.angularjs.org/guide/providers

Conexión con servicios

- **`$http`** Servicio básico para interactuar con servicios web

```
$http({ method: 'GET', url: '/api/locations'}).
    success(function (data, status, headers, config) {
      // data contiene la lista de localizaciones
    }).
   error(function (data, status, headers, config) {
      // la llamada asíncrona falló
   });
}
```

Las llamadas son asíncronas

Necesitamos un estandar para callbacks

Promesas $q

https://docs.angularjs.org/api/ng/service/$q

Formularios

- Angular mejora (mediante directivas) todos los elementos de formulario
- (o casi todos): input, select, textarea

La clave: ngModel

Conexión con servicios

#07
#08

- **`$http`** y promesas

  Parte de ES6. Es un estándar adoptado por otras plataformas (.Net, J2EE)

  https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Global_Objects/Promise

Es una buena práctica que todos nuestros servicios devuelvan promesas
(salvo los que sean explicitamente síncronos)

Para esto está **$q**

Custom directives

Hemos hablado y empleado muchas directivas, pero, ¿qué son?

- Son componentes reutilizables con HTML y comportamiento asociados.
- Pueden ser nuevos elementos o añadir comportamiento a los existentes
- AngularJS los reconoce en su proceso de compilación de HTML

Custom directives

Anatomía de una directiva

- Una directiva es una función que devuelve una estructura concreta:

```
export function DataDirective() : ng.IDirective {
    return {
        restrict: 'EAC',
        templateUrl: 'dashboard/todoDirective.html',
        replace: true,
        scope: {
        },
        controller: TodoDirectiveController,
        controllerAs: 'data',
        link: (scope, element, attributes) => {

        }
    }
}
```

Custom directives

http://github.com/jiparis/angular-checkbox-switch

Anatomía de una directiva

- Unifica en un sólo componente la plantilla y el controlador

```
export function DataTableDirective() : ng.IDirective {
    return {
        restrict: 'A',
        templateUrl: 'dashboard/dataTableDirective.html',
        scope: {
          customer: '=' // 2 way
           title: '@', // literal
          callback: '&' // one way (functions)
        },
        transclude: true,
        replace: true,
        controller: DataDirectiveController,
        require: 'ngModel',
        link: (scope, element, attributes, ctrl) => {

        }
    }
}
```

Url de la plantilla

Controlador

```
angular.directive('dataTable', DataDirective);
```

```
<data-table editMode=”edit” />
```

Estructura de la SPA

ngInclude

- Permite particionar la página en trozos independientes
- <ng-include src=””> o
- Src puede bindearse y ser dinámico (aunque para eso esta ng-route)

Estructura de la SPA

ngRoute

- Permite especificar rutas lógicas dentro de la aplicación
- ngView para pintar las vistas parciales

Formularios

ngModel

- Añade clases CSS según el estado de cada control (se ve inspeccionando con Chrome)

```
ng-valid ng-invalid ng-valid-[validacion] ng-invalid-[validacion] ng-
        pristine ng-dirty ng-touched ng-untouched ng-pending
```

Formularios

`ngModel`

A nivel de `<form name="theForm">`, se asigna un `FormContoller`, y se publica en el $scope: $scope.theForm…

Añade métodos:

- `$rollbackViewValue()`
- `$commitViewValue()`
- `$setValidity()`
- `$setDirty()`
- `$setPristine()`
- `$setUntouched()`
- `$setSubmitted()`

Añade propiedades:

- `$pristine`
- `$dirty`
- `$valid`
- `$invalid`
- `$submitted`
- `$error`

Luego los veremos

https://docs.angularjs.org/api/ng/type/form.FormController

Formularios

ngModel

Antes de continuar

ngModel realiza un 2-way binding, por lo que el modelo se actualiza inmediatamente

... Siempre y cuando el formulario sea válido

... pero si no es válido ...

danger → Undefined !!! ← danger

Consejo: bindear a una copia, y actualizar al salvar

Formularios

`ngModel`

A nivel de control `<form name="myForm"><input name="myInput">` añade un NgModelController, y se publica en el $scope: myForm.myInput … :

Añade métodos:

- `$render()`
- `$isEmpty()`
- `$setValidity()`
- `$setPristine()`
- `$setUntouched()`
- `$setTouched()`
- `$rollbackViewValue()`
- `$validate()`
- `$commitViewValue()`
- `$setViewValue()`

Añade propiedades:

- `$viewValue`
- `$modelValue`
- `$parsers`
- `$formatters`
- `$validators`
- `$asyncValidators`
- `$viewChangeListeners`
- **`$error`**
- `$pending`
- `$untouched`
- `$touched`
- `$pristine`
- `$dirty`
- `$valid`
- `$invalid`
- `$name`

Formularios

NgModel: opciones

- **updateOn**: por defecto 'change'
- **debounce**: timeout para aplicar y agrupar los cambios

Mostrar mensajes de error (ver nuevo message service)
Habilitar o deshabilitar botones

Formularios

Validaciones soportadas

`text, number, url, email, radio, checkbox, datetime-local, month, week, select, textarea`
`required, pattern, minlength, maxlength, min, max`

Errores en `myForm.myControl.$error.VALIDACION`

Directivas y ngModel:

- **Custom validators**.
  `NgModelCtrl.$validators.myValidator = (modelValue, viewValue):boolean;`
- Modificar validadores existentes
- **Custom controls**:
  `$setViewValue(...)` ← actualiza el modelo.
  `$render = (){elm.html(...)}` ← actualiza la vista

Formularios

Directivas externas útiles para formularios:

- Angular-xeditable

http://vitalets.github.io/angular-xeditable

Unit Testing

- Usamos **Jasmine**: `http://jasmine.github.io`
- Mocks AngularJS: angular-mocks.js

Incluir **DESPUES** de Jasmine

```
1  describe("Login controller", () => {
       ...

       // load the module
2      beforeEach(angular.mock.module("myApp"));

       beforeEach(inject((
           $httpBackend,
3          $rootScope,
           ...) => {

           cnt = new MyApp03.LoginController($rootScope, loginService, $log, $location,
$timeout);
           ...
       }));

       afterEach(function () {
4          backend.verifyNoOutstandingExpectation(); // HTTP
           backend.verifyNoOutstandingRequest();  // HTTP
       });
```

1. Crea la suite
2. Carga el módulo (mock)
3. Prepara los tests (crea el scope, instancia controllers, …)
4. Tras la ejecución de tests, verifica que no hay peticiones pendientes o ausentes

Unit Testing

- Usamos **Jasmine:** `http://jasmine.github.io`

```
it("should change location on success", () => {
    backend.whenPOST("/api/login").respond(200);     // OK

    cnt.doLogin("user", "pwd");

    scope.$digest();
    backend.flush();

    expect(location.path()).toBe("/dashboard");
});
```

1. mock de backend
2. Emula ciclo de $digest (llama a $watch ...)
3. ejecuta requests
4. expectations

Testing

Automatización con karma

```
0. Instalar nodejs
1. npm init
2. npm install karma-cli -g
3. npm install karma --save-dev
4. npm install karma-jasmine karma-coverage --save-dev
5. npm install karma-chrome-launcher --save-dev
6. npm install karma-phantomjs-launcher --save-dev
```

o npm install si ya tenemos el packages.json

```
karma init karma.config.js
Editar karma.config.js (rutas, reporters, ...)
karma start karma.config.js
```

Browsers: PhantomJS, Chrome ...

Reporters: progress, teamcity, coverage, html, mocha ...

Preprocessors: coverage ...