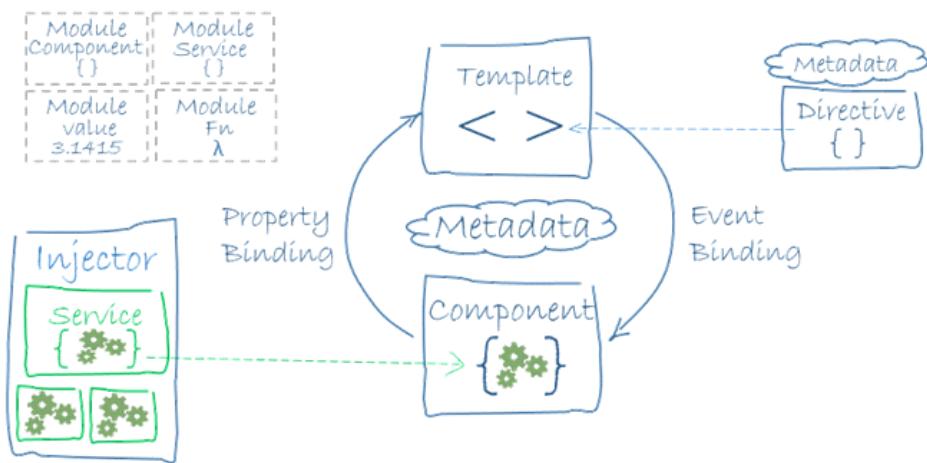


Ξ Angular. Segunda Parte

lunes, 11 de septiembre de 2017 22:13

Arquitectura Angular



Elementos principales

Modules
Components
Templates
Metadata
Data binding
Directives
Services
Dependency injection

<https://angular.io/docs/ts/latest/guide/architecture.html>

Guía de estilo

sábado, 14 de octubre de 2017 20:13

<https://angular.io/guide/styleguide>

Getting Started
Tutorial
Fundamentals
Techniques
Internationalization (i18n)
Security
Setup & Deployment
Upgrading
Visual Studio 2015 QuickStart
Style Guide
Glossary
API
stable (v4.4.4)

Style Guide

Looking for an opinionated guide to Angular syntax, conventions, and application structure? Step right in! This style guide presents preferred conventions and, as importantly, explains why.

Style vocabulary

Each guideline describes either a good or bad practice, and all have a consistent presentation.

The wording of each guideline indicates how strong the recommendation is.

Do is one that should always be followed. Always might be a bit too strong of a word. Guidelines that literally should always be followed are extremely rare. On the other hand, you need a really unusual case for breaking a *Do* guideline.

Consider guidelines should generally be followed. If you fully understand the meaning behind the guideline and have a good reason to deviate, then do so. Please strive to be consistent.

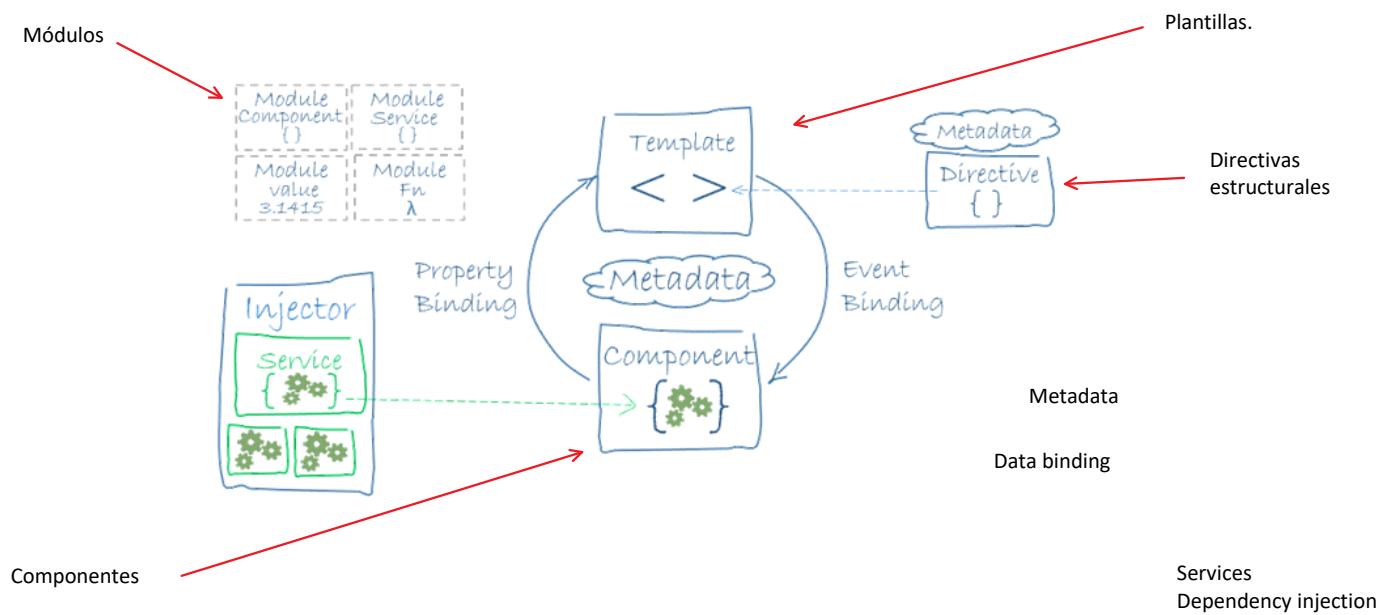
Avoid indicates something you should almost never do. Code examples to avoid have an unmistakable red header.

Why? gives reasons for following the previous recommendations.

- *Style vocabulary*
- *File structure conventions*
- *Single responsibility*
 - *Rule of One*
 - *Small functions*
- *Naming*
 - *General Naming Guidelines*
 - *Separate file names with dots and dashes*
 - *Symbols and file names*
 - *Service names*
 - *Bootstrapping*
 - *Directive selectors*
 - *Custom prefix for components*
 - *Custom prefix for directives*
 - *Pipe names*
 - *Unit test file names*
 - *End-to-End (E2E) test file names*
 - *Angular NgModule names*
- *Coding conventions*
 - *Classes*
 - *Constants*
 - *Interfaces*
 - *Properties and methods*
 - *Import line spacing*
- *Application structure and NgModules*
 - *LIFT*
 - *Locate*
 - *Identify*
 - *Flat*
 - *T-DRY (Try to be DRY)*
 - *Overall structural guidelines*
 - *Folders-by-feature structure*
 - *App root module*
 - *Feature modules*
 - *Shared feature module*
 - *Core feature module*
 - *Prevent re-import of the core module*
 - *Lazy Loaded folders*
 - *Never directly import lazy loaded folders*
- *Components*
 - *Component selector names*
 - *Components as elements*
 - *Extract templates and styles to their own files*
 - *Decorate input and output properties*
 - *Avoid aliasing inputs and outputs*
 - *Member sequence*
 - *Delegate complex component logic to services*
 - *Don't prefix output properties*
 - *Put presentation logic in the component class*
- *Directives*
 - *Use directives to enhance an element*
 - *HostListener/HostBinding decorators versus host metadata*
- *Services*
 - *Services are singletons*
 - *Single responsibility*
 - *Providing a service*
 - *Use the @Injectable() class decorator*
- *Data Services*
 - *Talk to the server through a service*
- *Lifecycle hooks*
 - *Implement lifecycle hook interfaces*
- *Appendix*
 - *Codelyzer*
 - *File templates and snippets*

Módulos. Componentes. Vistas

lunes, 11 de septiembre de 2017 22:17



Módulos

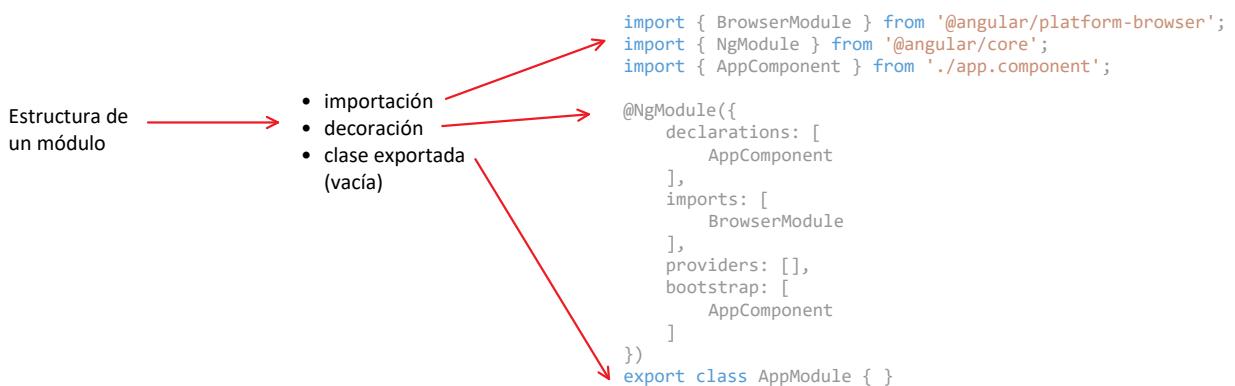
lunes, 11 de septiembre de 2017 22:23

Agrupación de componentes y otros elementos que son declarados en la decoración de una determinada clase

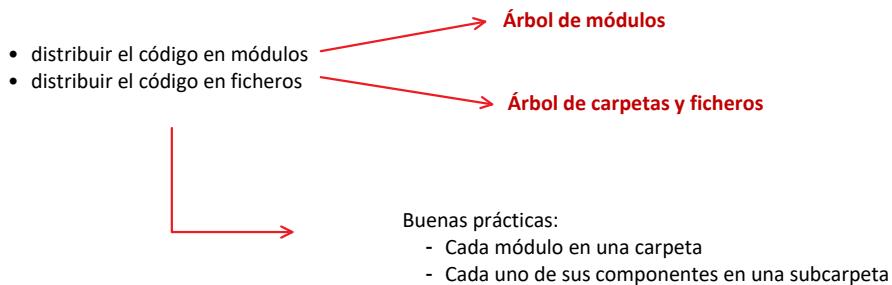
Toda app tiene

al menos un módulo que define los componentes de la app : AppModule

incluyendo al menos el componente principal AppComponent -> selector: app-root



En cualquier aplicación hay que diferenciar 2 procesos



Independientemente de ambos, el uso de los componentes en el código HTML genera un **árbol de componentes**

Ejemplo: Módulos

- app
 - o clientes
 - o proveedores
 - o productos
 - o *shared*

Módulos funcionales

Módulo compartido

Importación

Lunes, 11 de septiembre de 2017 22:32

Todos los ficheros de *typescript/JS* que tengan que utilizarse dentro del módulo tienen que ser importados



Desaparecen los <script> en el HTML durante el desarrollo

No se indica la extensión:

- en tiempo de desarrollo será TS
- en ejecución será JS, una vez *transpilado* y se reagrupará en el bundle.js

El comando *import* siempre implica dos elementos

```
import{ nombre de la clase }from 'donde esta la clase';
```

Si se omite el nombre de la clase {*} -> se importaran todas las que existan en el sitio indicado
(No es recomendable, por cómo funciona el *tree shaking*).

Dos posibles "orígenes" a la hora de importar elementos

por **nombre simbólico**, desde *node_modules*, los elementos de Angular y otras librerías

```
import{ NgModule }from '@angular/core';
```

por **ruta**, desde un *path*, relativo a nuestra "base", los elementos de la aplicación
(esa "base" se define en los metadatos de index.html)

```
import { AppComponent }from './app.component';
```

Un módulo también puede ser importado desde otro, dando lugar a un árbol de módulos

Importación: index.ts



En TypeScript se pueden definir ficheros index.ts que exportan todos los ficheros de una carpeta.

- simplifica la importación desde otros ficheros
- desacopla los tipos del fichero en el que se declaran

index.ts

```
export * from './app.component';
export * from './app.module';
```

Fichero que lista todos los ficheros TS de una carpeta

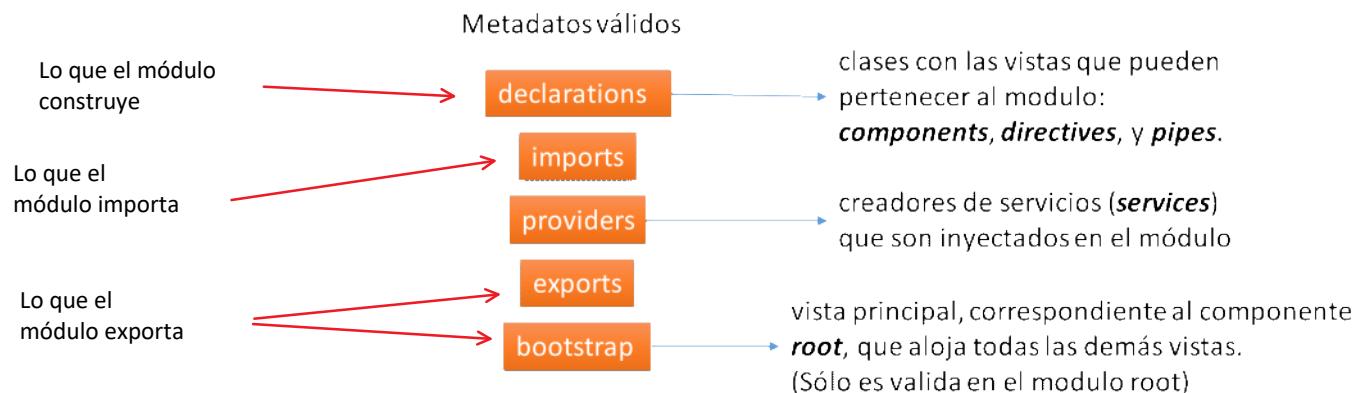
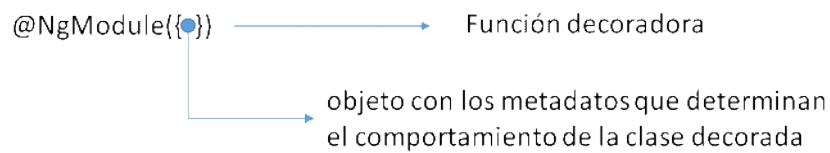
src/main.ts

```
import { AppModule } from './app/';
```

Al importar ese fichero se puede referenciar cualquier clase de la carpeta (similar a paquetes Java)

Decoración / Anotación

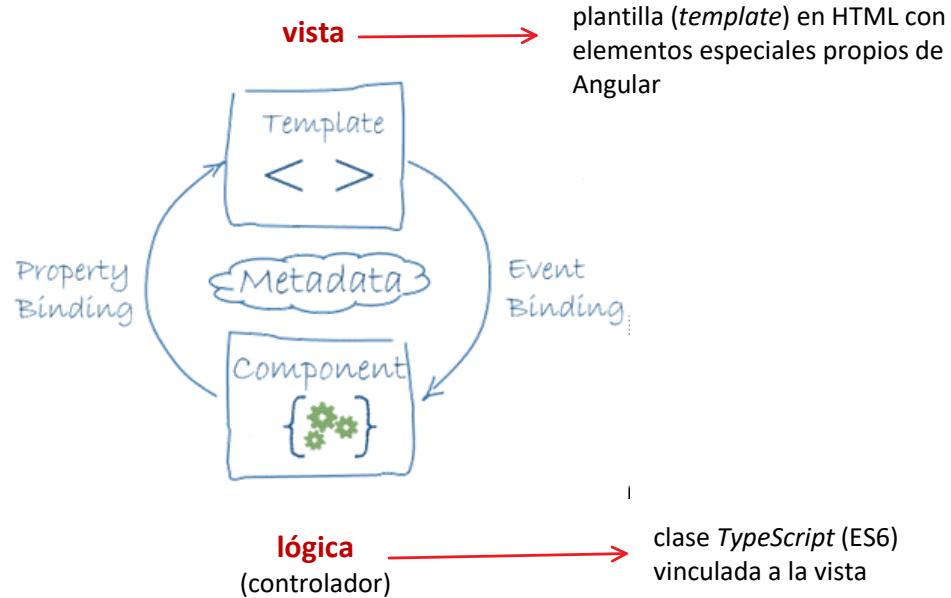
Lunes, 11 de septiembre de 2017 22:49



Componentes

lunes, 11 de septiembre de 2017 22:52

Un componente supone una nueva **etiqueta HTML** con una vista y una lógica definidas por el desarrollador



Estructura de un componente

sábado, 14 de octubre de 2017 20:30

Estructura de
un componente

- ▶ importación
- ▶ decoración
 - ▶ selector
 - ▶ template / templateUrl
 - ▶ ...
- ▶ clase exportada

Importamos al menos la
clase *Component* de Angular

```
import { Component } from '@angular/core';
```

Dicha clase puede usarse
en forma de decorador

```
@Component({  
  selector: 'app-root',  
  templateUrl: './app.component.html',  
  styleUrls: ['./app.component.css']  
)  
export class AppComponent {  
  ...  
}
```

Este decorador
concreto admite
determinados
metadatos

Nombre mediante el que podrá ser
importada para que forme parte de
un módulo
(suele coincidir con el nombre del
fichero)

```
@Component({  
  selector:  
  template:  
  templateUrl:  
  styles:  
  styleUrls:  
  encapsulation:  
  animations:  
})
```

Otros metadatos

```
changeDetection  
viewProviders  
moduleId  
interpolation  
entryComponents  
preserveWhitespaces  
  
// inherited from core/Directive  
host  
providers  
exportAs  
queries
```

Ya no se utilizan los metadatos

```
directives  
pipes  
inputs  
outputs
```

<https://angular.io/api/core/Component>

Vistas HTML

lunes, 11 de septiembre de 2017 22:54

La **vista** del componente (**HTML**) se genera en función de dos elementos

- su estado, definido por el valor de los **atributos de la clase** en un determinado momento
- la plantilla o *template* que tiene asociado el componente, donde además de HTML puede haber referencia a dichos atributos

```
export class AppComponent{  
    name = 'Curso de Angular';  
    imgUrl = "assets/logo.jpg";  
}
```

Propiedades de la clase

Esa relación se denomina "*binding*"

```
<h1>Hola {{name}}!</h1>  
<img [src]="imgUrl"/>
```

Uso de esas propiedades desde la vista

Recursos de la aplicación

lunes, 11 de septiembre de 2017 23:19

Los recursos (imágenes, fonts..) deben colocarse en una carpeta src/assets para que se copien en el build



Creación de componentes

lunes, 11 de septiembre de 2017 23:21

Utilizamos angular-cli para generar la base de un nuevo componente

```
ng g(enerate) c(omponent) "nombre"
```

Se habrá añadido directamente en el módulo

- como import
- como declaration
- Dispondremos de una carpeta con la estructura de archivos.

Inicialmente crearemos un componente "dumpy" :
stateless, incluso sin lógica ninguna.
Lo incorporaremos al componente principal

Cuando utilizamos angular-cli para generar la base de un nuevo componente,
la clase se crea con una estructura de partida, que es la recomendada
en todos los componentes

```
export class <nombre> implements OnInit {
    constructor() {} : inyección de dependencias
    ngOnInit() {} : inicialización de valores

}

export class FeatureComponent implements OnInit {
    constructor() { }

    ngOnInit() [ ]
}
```

Ejemplo: creación de 1 componente

Junes, 11 de septiembre de 2017 23:22

app-root

Hola Mundo 2 componentes

app-pie

Creamos una nueva aplicación

ng new hola-componentes

Modificamos el componente principal como en casos anteriores

Creamos un nuevo componente

ng g c pie

En el módulo principal se añade

```
import { PieComponent } from './pie/pie.component';
@NgModule({
  declarations: [
    AppComponent,
    PieComponent
  ],
  ...
})
```

Cambiamos la lógica (*controller*) de nuestro componente

```
import { Component, OnInit } from '@angular/core';
@Component({
  selector: 'app-pie',
  templateUrl: './pie.component.html',
  styleUrls: ['./pie.component.css']
})
export class PieComponent implements OnInit {
  public formador: string
  public empresa: string
  public fecha : string

  constructor() {}

  ngOnInit() {
    this.formador = "Alejandro Cerezo Lasne"
    this.empresa = "Icono Training Consulting"
    this.fecha = "2017"
  }
}
```

Cambiamos el contenido de la vista de nuestro componente

```
<footer>
  <p>{{formador}} - {{fecha}}</p>
  <p>{{empresa}}</p>
</footer>
```

Cambiamos el CSS específico de nuestro componente

```
footer {
  position: fixed;
  bottom : 0;
  width: 100%;
  border-top: 1px papayawhip solid
}
p {
  text-align: center;
  font-size: 1.3em;
  color : papayawhip;
}
```

Consumimos el componente <app-pie> desde la vista del componente principal

```
<header style="text-align:center">
  <h1>
    Bienvenidos al curso {{curso}}!
  </h1>
  
</header>
<app-pie></app-pie>
```

Bienvenidos al curso Angular 2.x!



Alejandro Cerezo Lasa - 2017
Icono Training Consulting

Versión Angular2 del
tradicional "Hola
Mundo" con un pie
creado como
componente
independiente

Ejercicio: Módulos y componentes

sábado, 23 de septiembre de 2017 20:39

Módulos y componentes

- App -> app-main
 - Core ->cabeza, pie



app.module.ts

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { AppComponent } from './app.component';
import { SharedModule } from './shared/shared.module';

@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
    SharedModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

El módulo *shared* es incluido / vinculado en el módulo principal

core.module.ts

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';
import { CabezaComponent } from './cabeza/cabeza.component';
import { PieComponent } from './pie/pie.component';
@NgModule({
  imports: [
    CommonModule
  ],
  declarations: [
    CabezaComponent,
    PieComponent],
  exports: [
    CabezaComponent,
    PieComponent
  ]
})
export class SharedModule { }
```

Los componentes del módulo *shared* son declarados y referenciados como exportables

Ciclo de vida de los componentes

miércoles, 13 de septiembre de 2017 22:20

I'm Todd, a Developer Advocate @Telerik. Founder of @UltimateAngular Creator of the ngMigrate. JavaScript, Angular, React, conference speaker. Developer Expert at Google.

Master Angular 1.x and Angular 2 with me online

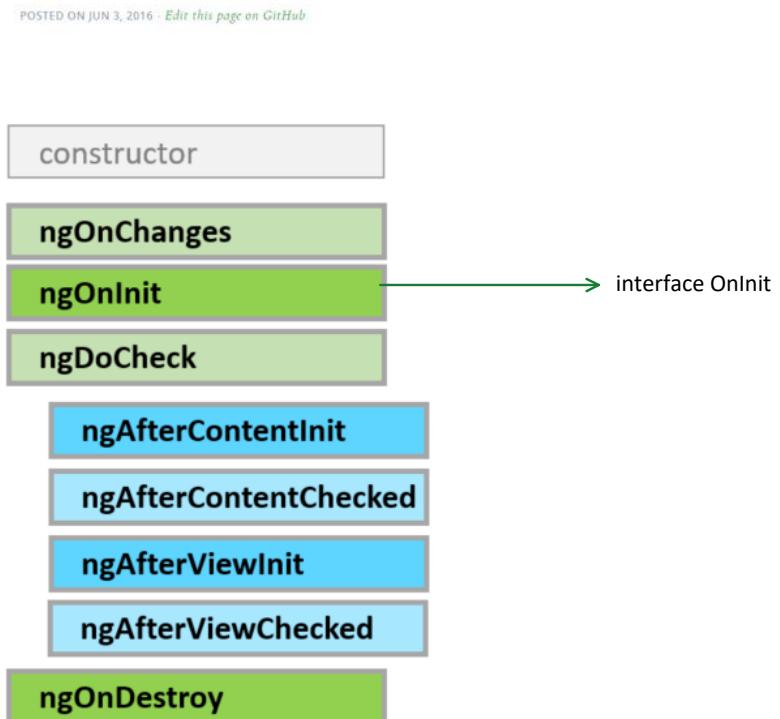
Limited Angular 2 preorders now available.
Master the latest Angular 1.5 components, or preorder the most in-depth Angular 2 courses.

See the courses >

Implementando en Angular 1.5

Lifecycle hooks in Angular 1.5

<https://toddmotto.com/angular-1-5-lifecycle-hooks>



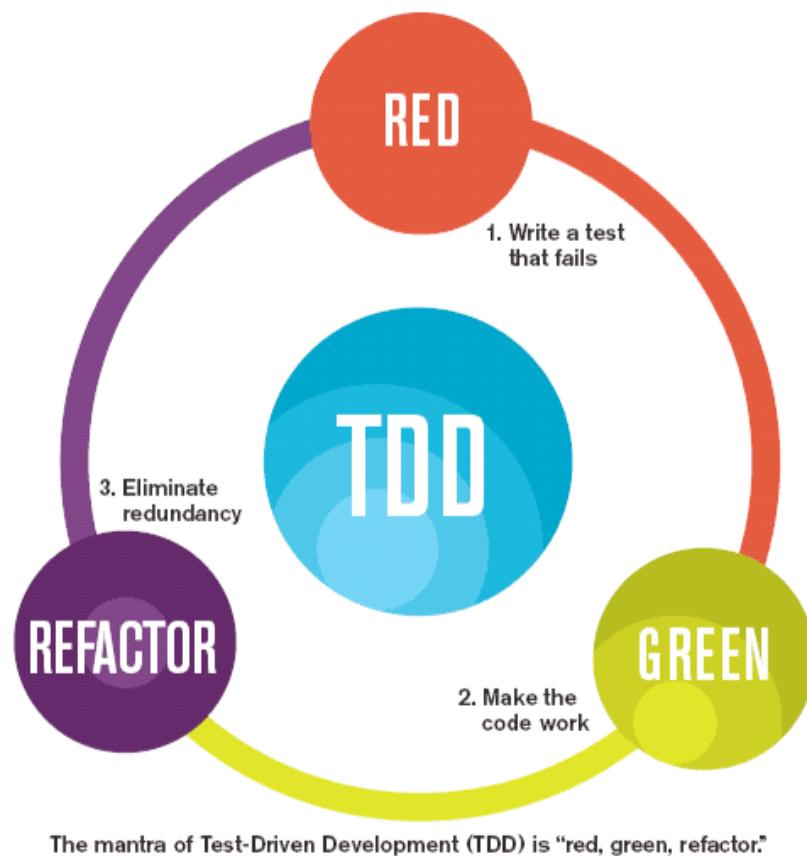
<https://angular.io/docs/ts/latest/guide/lifecycle-hooks.html>

Entorno de Testing

viernes, 26 de enero de 2018 17:46

Cuando se definen las pruebas

- antes del desarrollo (*TDD - Test Driven Development*)
conceptualmente mucho más complejo
una de las propuestas ligadas a las metodologías ágiles, junto con el *refactoring*
- después del desarrollo



Tipos de pruebas

domingo, 11 de febrero de 2018 17:49

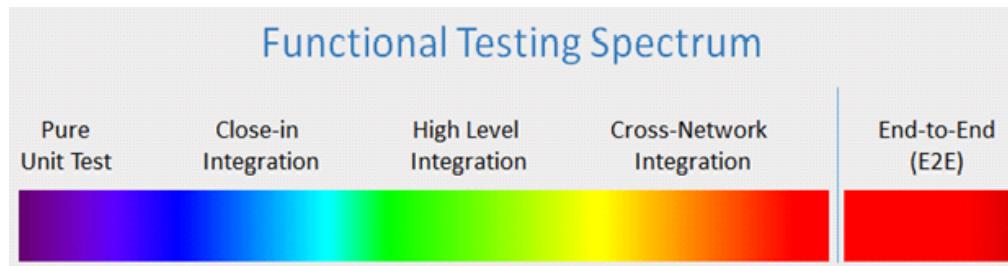
Pruebas unitarias
- (en desarrollo front)

Se critica que la mejora obtenida en la calidad no se compensa con el esfuerzo de usar y mantener un entorno de pruebas (*testing*)

De alguna manera Angular facilita las pruebas para desmentir esta crítica, proporcionando un entorno de pruebas ya pre-configureo: **Karma**, un ejecutador de pruebas, que se programan en **Jasmine**

Pruebas e2e

Son de gran utilidad:
permiten simular de forma automatizada las interacciones del usuario



Herramientas de terceros incluidas en Angular-cli



Del arranque de *Karma* y *Protractor* se ocupan los comandos (scripts) de npm

- *Karma*: se ocupa el comando de **npm test** (se ejecuta directamente igual que **npm start**)
- *Protractor* se inicia con el comando **e2e**, que al no ser estándar se inicia con **npm run e2e**

Pruebas unitarias

viernes, 26 de enero de 2018 18:13

Definición: Aspectos generales de Jasmine

Ficheros `.spec` creados automáticamente por el cli para los componentes y servicios utilizando la sintaxis de Jasmine

```
describe ("Descripción de la suite de pruebas", function ()  
{  
    it ("Descripción de la prueba 1, function(): boolean {}")  
    it ("Descripción de la prueba 2, function(): boolean {}")  
})
```



En las funciones `it` se utiliza el método `expect("expresión")` que define las expectativas de la prueba sobre una determinada expresión o variable

al resultado puede concatenársele la definición de la evaluación a realizar gracias al conjunto de métodos soportados en el *framework* Jasmine

```
expect(variable).toEqual(valorEsperado)
```

Previamente al `it`, para preparar la prueba se utiliza el método `beforeEach`, que recibe una función que se encargará de preparar las pruebas

```
beforeEach(function() {})
```

Ejemplo

```
import { async, ComponentFixture, TestBed } from '@angular/core/testing';  
import { PieComponent } from './pie.component';  
  
Suite de pruebas → describe('PieComponent', () => {  
    let component: PieComponent;  
    let fixture: ComponentFixture<PieComponent>;  
  
    beforeEach(async(() => {  
        TestBed.configureTestingModule({  
            declarations: [ PieComponent ]  
        }).compileComponents();  
    }));  
  
    beforeEach(() => {  
        fixture = TestBed.createComponent(PieComponent);  
        component = fixture.componentInstance;  
        fixture.detectChanges();  
    });  
  
    Pruebas it de diversas funcionalidades → it('should create', () => {  
        expect(component).toBeTruthy();  
    });  
  
    Preparación del "lecho" en el que se instanciará el componente →  
    Configuración del módulo de pruebas →  
    Configuración del componente →
```

Pruebas *it* incluida por defecto en todos los componentes
se limita a comprobar que el componente existe

Elementos específicos de Angular

viernes, 26 de enero de 2018 18:36

TestBend -> Lecho o "camilla" de la prueba: un módulo de pruebas, equivalente a cualquier módulo, pero específico para las pruebas

En un *BeforeEach* se incluye la configuración de este módulo, gracias al método *configureTestingModule()*

Para completar la pre-configuration de Angular suele ser necesario que el módulo de pruebas sea muy similar al módulo real del componente

En un fichero *spec* de un componente existirán dos variables:

- *component* del tipo del componente asociado
- *fixture* (accesorios) del tipo *ComponentFixture* y el genérico (subtipo) del componente asociado

```
let component: MiComponent;
let fixture: ComponentFixture<MiComponent>;
```

En un segundo *BeforeEach* se instancian

```
beforeEach(() => {
  fixture = TestBed.createComponent(TestComponent);
  component = fixture.componentInstance;
  fixture.detectChanges();
});
```

la fixture, creando en el *TestBend* un componente de la clase que estamos probando

el componente en sí mismo a partir de la fixture

Estas fixture son una forma de manipular la creación de un componente, por ejemplo dando valor a las propiedades de tipo input del componente

Creación de pruebas específicas

domingo, 28 de enero de 2018 17:01

Acceso a un elemento del DOM

Existe un componente con la siguiente plantilla

```
template: `<p id="test" destacar></p>`
```

Gracias a la propiedades de la fixture se puede acceder al elemento del DOM que será renderizado partir de la plantilla

```
import { By } from '@angular/platform-browser';
import { DebugElement } from '@angular/core';

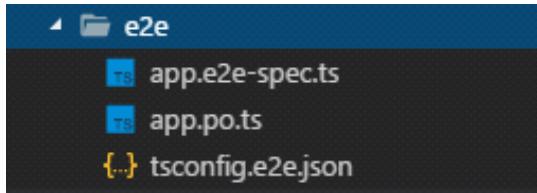
let elem: DebugElement;
elem = fixture.debugElement.query(By.css('#test'));
```

Existen varias opciones para *By* siendo la más común utilizar selectores CSS:

- id
- selector de etiqueta
- selector de atributo
- pseudoclases de posición...

Pruebas e2e

domingo, 28 de enero de 2018 20:53



Se agrupan todas en la carpeta e2e en la raíz del proyecto

- fichero de expectativas
- fichero de pruebas
- fichero de configuración

Muy similar al que utiliza Karma:

- suite de pruebas
- preparación: define la página a probar
- funciones it: en cada caso de invoca una función definida en el fichero de pruebas

Define las pruebas concretas

- accede al DOM
- simula la interacción con el usuario

app.e2e-spec.ts

```
import { AppPage } from './app.po';
describe('base-app App', () => {

    let page: AppPage;

    beforeEach(() => {
        page = new AppPage();
    });

    it('should display app title', () => {
        page.navigateTo();
        expect(page.getTitle()).toEqual('Angular Avanzado!');
    });

    it('should display app footer', () => {
        page.navigateTo();
        expect(page.getFooter()).toBeTruthy();
    });
});
```

app.po.ts

```
import { browser, by, element } from 'protractor';
export class AppPage {

    navigateTo() {
        return browser.get('/');
    }
    getTitle() {
        return element(by.css('app-root h1')).getText();
    }
    getFooter() {
        return element(by.css('app-root footer')).getText();
    }
}
```

Plantillas (Templates)

Las plantillas (*templates*) permiten definir la vista en función de la información del componente

Desarrollo declarativo

En todas las directivas es posible añadir el prefijo data- para que las directivas no supongan problema de validación

expansión de las características del HTML, añadiéndole funcionalidades sin necesidad de escribir código JavaScript

Se puede ver como una forma de agregar valor semántico al HTML.

Lenguaje de plantillas

```
 {{user.name}}  
 href = {{miUrl}}  
 [href] = "miUrl"  
 (click)="usarBoton()"
```

[] a cualquier atributo HTML se le asigna una variable del componente

() a cualquier evento HTML se le asigna un método del componente

{()} se interpola una variable

[()] two way binding

se declara una variable local en la vista

- Expresiones
- Directivas estructurales
 - Visualización condicional
 - Repetición de elementos
- Directivas y Estilos CSS

lenguaje de plantillas mediante {()}

atributos *ng-específicos de angular que se pueden asignar a etiquetas HTML.

atributos ng que gestionan dinámicamente los estilos y clases CSS



Formularios

<https://angular.io/docs/ts/latest/guide/template-syntax.html>

Eventos

domingo, 24 de septiembre de 2017 11:57

Otro elemento clave para entender el funcionamiento de las vistas son los eventos del sistema

- proporcionados por el navegador, como interfaz con el S.O. y
- definidos en HTML5

Eventos del sistema (1)



Evento	Descripción	Elementos para los que está definido
blur	Deseleccionar el elemento	<button>, <input>, <label>, <select>, <textarea>, <body>
change	Deseleccionar un elemento que se ha modificado	<input>, <select>, <textarea>
click	Pinchar y soltar el ratón	Todos los elementos
dblclick	Pinchar dos veces seguidas con el ratón	Todos los elementos
focus	Seleccionar un elemento	<button>, <input>, <label>, <select>, <textarea>, <body>
keydown	Pulsar una tecla (sin soltar)	Elementos de formulario y <body>
keypress	Pulsar una tecla	Elementos de formulario y <body>
keyup	Soltar una tecla pulsada	Elementos de formulario y <body>
load	La página se ha cargado completamente	<body>



Eventos del sistema (2)

Evento	Descripción	Elementos para los que está definido
mousedown	Pulsar (sin soltar) un botón del ratón	Todos los elementos
mousemove	Mover el ratón	Todos los elementos
mouseout	El ratón "sale" del elemento (pasa por encima de otro elemento)	Todos los elementos
mouseover	El ratón "entra" en el elemento (pasa por encima del elemento)	Todos los elementos
mouseup	Soltar el botón que estaba pulsado en el ratón	Todos los elementos
reset	Inicializar el formulario (borrar todos sus datos)	<form>
resize	Se ha modificado el tamaño de la ventana del navegador	<body>
select	Seleccionar un texto	<input>, <textarea>
submit	Enviar el formulario	<form>
unload	Se abandona la página (por ejemplo al cerrar el navegador)	<body>

Nuevos eventos en HTML5



Window

onafterprint
onbeforeprint
onbeforeunload
onerror
onhaschange
onmessage
onoffline
ononline
onpagehide
onpageshow
onpopstate
onredo
onresize
onstorage
onundo

Form

oncontextmenu
onformchange
onforminput
oninput
oninvalid

Mouse

ondrag
ondragend
ondragenter
ondragleave
ondragover
ondragstart
ondrop
onmousewheel
onscroll

Media Events

oncanplay
oncanplaythrough
ondurationchange
onemptied
onended
onerror
onloadeddata
onloadedmetadata
onloadstart
onpause
onplay
onplaying
onprogress

onratechange
onreadystatechange
onseeked
onseeking
onstalled
onsuspend
ontimeupdate
onvolumechange
onwaiting

Gestión de eventos

domingo, 15 de octubre de 2017 9:36

El operador () indicando su nombre dentro de los paréntesis, permite definir el manejador de cualquier evento estándar de un determinado elemento del DOM.

El objeto **\$event**, muy similar al utilizado en JQuery, se envía como parámetro al manejador de evento que se defina

```
(click) = "btnResponder($event)"  
  
↓  
btnResponder (oEvent) {  
    ... // respuesta al evento  
    console.log(oEvent)  
}
```

- aunque no es una práctica recomendada, puede ser una expresión asociada al evento
(click) = "++nCount"
- una llamada a una función definida en la clase que constituye el componente
(click) = "setContador(2)"

```
▼ MouseEvent {isTrusted: true, screenX: 2232, screenY: 592  
altKey: false  
bubbles: true  
button: 0  
buttons: 0  
cancelBubble: false  
cancelable: true  
clientX: 493  
clientY: 401  
composed: true  
ctrlKey: false  
currentTarget: null  
defaultPrevented: false  
detail: 1  
eventPhase: 0  
fromElement: null  
isTrusted: true  
layerX: 493  
layerY: 401  
metaKey: false  
movementX: 0  
movementY: 0  
offsetX: 24  
offsetY: 8  
pageX: 493  
pageY: 401  
► path: (9) [button, form.ng-valid.ng-dirty.ng-touched,  
relatedTarget: null  
returnValue: true  
screenX: 2232  
screenY: 592  
shiftKey: false  
► sourceCapabilities: InputDeviceCapabilities {firesTouchEvents: true}  
► srcElement: button  
► target: button  
timeStamp: 196737.77000000002  
► toElement: button
```

Propiedades y Directivas

lunes, 2 de octubre de 2017 21:59

Angular permite manipular las propiedades del DOM, aunque aparentemente haga referencia a los atributos HTML.

Hay que recordar que los atributos HTML suelen reflejarse en las correspondiente propiedades del DOM, pero en algunos casos esta relación no es tan directa.

En AngularJS las propiedades de los elementos del DOM se manipulaban mediante directivas



Existía un gran número de directivas

A partir de Angular 2 se utiliza el operador [] que asocia cualquier propiedad de un elemento con una variable del modelo



existen muy pocas directivas

Ejemplos

Directivas y referencias



[src]

indica la *url* del fichero que actúa como fuente para una etiqueta **img**, sustituyendo el habitual atributo *src*

En lugar de

La directiva cumple una función similar a *ng-bind* cuando se emplean expresiones, evitando que se muestre la expresión o un ícono antes de que haya sido cargada la imagen.

[href]

cumple una función similar respecto a los hiperenlaces (etiqueta **a**), evitando que se pueda pinchar en uno de ellos con una expresión de AngularJS antes de que ésta se haya resuelto.

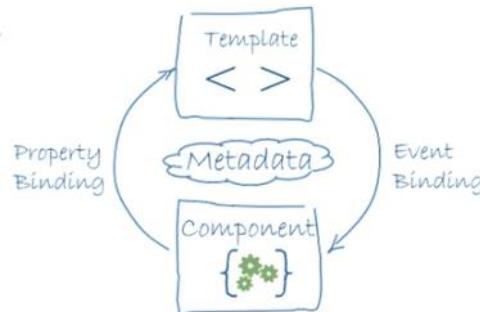
Lógica básica: binding

domingo, 15 de octubre de 2017 8:22

Angular permite manipular las propiedades del DOM, aunque aparentemente haga referencia a los atributos HTML.

Este mecanismo se conoce como **acceso a las propiedades (Input Property Binding)** y utiliza el formato []

Permite cambiar los valores pero no conocer cuando se producen esos cambios.



La vista es capaz de informar de los cambios que se producen, i. e. eventos

Este mecanismo se conoce como **respuesta a eventos (Output Event Binding)** y utiliza el formato (<evento>)

La combinación del acceso a la propiedad *value*, con el evento correspondiente al cambio de valor en esa propiedad se conoce como **doble binding (two-way data binding)**

En la práctica

Cambiando
- directivas,
- eventos y
- expresiones
tenemos:

Acceso a datos del modelo (*Property binding*)
<input type="text" [ngModel]="name">

Respuesta a eventos (*Event Binding*)
<button (click)="setName('Pepe')">

Datos enlazados (*two-way data binding*)
<input type="text" [(ngModel)]="name">
{name}

Ejemplo

```
<input type="text" id="nombre" name="nombre"
[value]= "sNombre">
<button (click)="sNombre=''">Borrar</button>
<p>Hola {{sNombre}}</p>
```

En el controlador

```
this.nombre = 'Pepe';
```

Usando Bindings básicos

Dime tu nombre

Hola Pepe

El valor inicial de la propiedad se establece accediendo a la variable sNombre del controlador

En respuesta al evento clic se modifica el valor de la variable sNombre.

Una expresión refleja el valor de la variable sNombre.

Usando Bindings básicos

Dime tu nombre

Hola

Al modificar la variable sNombre se refleja en la vista

Usando Bindings básicos

Dime tu nombre

Hola Pepe

Al modificar la vista NO se refleja en la variable del modelo/controlador

en la vista

variable del modelo/controlador

Se trata de un binding en una sola dirección

ngModel y doble binding

domingo, 24 de septiembre de 2017 11:11

Directiva ngModel



relaciona elementos del DOM con modelos de datos, informando al compilador HTML para que tome una variable del modelo.

- Se utiliza en los controles de formulario, como INPUT, SELECT, TEXTAREA o controles personalizados.
- la notación [] enlaza (*binding*) una **propiedad** de la clase que define al componente con el correspondiente control de formulario de la vista

```
<input type="text" [ngModel]="name">
```

además de su funcionamiento normal, en una dirección, cuando se invoca como propiedad, este "enlace" puede funcionar en las dos, cuando se combina con el uso de eventos ()

En realidad, la directiva *ngModel* está agrupando 2 operaciones

```
<input type="text" id="nombre"
       [value] = "sNombre"
       (input) = 'sNombre = $event.target.value'>
```

Doble binding



combinación de

- la directiva *ngModel*
- el evento de Angular *ngModelChange*

```
<input type="text" id="nombre"
       [ngModel] = "sNombre"
       (ngModelChange) = 'sNombre = $event'>
```

Al no ser un evento del sistema, cambia el valor de \$event

Forma abreviada de escribirlo [...]



"banana in a box"

```
<input type="text" id="nombre"
       [(ngModel)] = "sNombre">
```

Sirve de base al doble binding

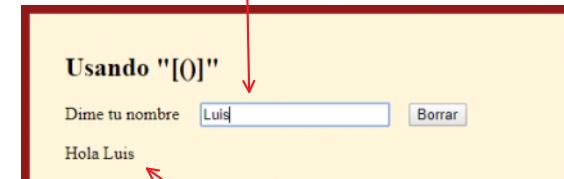
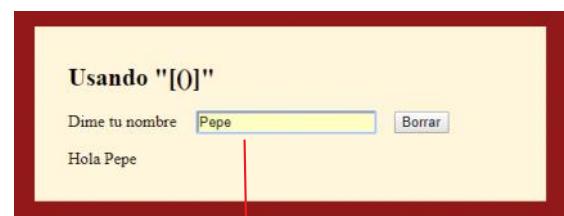
<https://angular.io/guide/template-syntax>

Ejemplo

```
<div>
  <h2>Usando "[()]"</h2>
  <form>
    <label for="nombre">Dime tu nombre </label>
    <input type="text" id="nombre" name="nombre"
           [(ngModel)] = "nombre">
    <button (click)='btnBorrar($event)'>Borrar</button>
  </form>
  <p>Hola {{nombre}}</p>
</div>
```

```
import { Component, OnInit } from '@angular/core';

@Component({
  selector: 'app-formulario-bnb',
  templateUrl: './formulario.component.html',
  styleUrls: ['./formulario.component.css']
})
export class FormularioBnbComponent implements OnInit {
```



```

    templateUrl: './formulario.component.html',
    styleUrls: ['./formulario.component.css']
})
export class FormularioBnbComponent implements OnInit {
  public nombre: string;
  constructor() { }
  ngOnInit() {
    this.nombre = '';
  }
  btnBorrar (evento) {
    this.nombre = '';
    console.log(evento);
  }
}

```

The screenshot shows a simple web interface. At the top, there is an input field with the placeholder "Dime tu nombre" and a text box containing the name "Luis". To the right of the input field is a button labeled "Borrar". Below the input field, the text "Hola Luis" is displayed. A red arrow points from the text "Hola Luis" back up towards the code snippet above, indicating a connection between the code's logic and the resulting user interface.

En este caso, al modificar la vista SI se refleja en la variable del modelo/controlador

Expresiones



- Lógica limitada: no se pueden incluir en ellas condicionales (excepto el operador ternario), bucles o excepciones
- Se les puede dar formato mediante filtros

Referencias a modelos

`{{Dato}}`

Operaciones
aritméticas básicas

`{{Dato + 4}}`

Empleo de los métodos de
los objetos envolventes de
JS (e.g. String)

`{{"Beginning AngularJS".toUpperCase()}}`
`{{"ABCDEFG".indexOf('D')}}`

Uso del operador ternario

`{{Dato == 1 ? "Red" : "Blue"}}`



Ejemplos de expresiones

Ejemplos de Expresiones

Operaciones aritméticas básicas

$6 + 4 = 10$

El resultado se obtiene con la expresión: `{(6 + 4)}`

Empleo de los métodos de los objetos envolventes de JS (e.g. String)

BEGINNING ANGULARJS

Resultado de la expresión: `{"Beginning AngularJS".toUpperCase()}`

3

Resultado de la expresión: `{"ABCDEFG".indexOf('D'))}`

Uso del operador ternario

Red

Resultado de la expresión: `{(1==1 ? "Red" : "Blue")}`

Alejandro L. Cerezo - Madrid 2015

```
nNumber: number;
sName: string;
isSpanish: boolean;
constructor() { }
ngOnInit() {
  this.nNumber = 22;
  this.sName = 'Pepe';
}

<div class="card mb-2">
  <div class="card-header">
    <p class="h4">Expresiones</p>
  </div>
  <div class="card-body">
    <div class="form-check">
      <label class="form-check-label">
        <input type="checkbox" class="form-check-input" name="" id=""
          [(ngModel)]="isSpanish">
          Selecciona si hablas español
      </label>
    </div>
  </div>
  <div class="card-footer text-muted">
    <p>Hola {{sName.toUpperCase()}}</p>
    <p>El producto de {{nNumber}} * 4 es: {{nNumber * 4}}</p>
    <p>{{isSpanish ? 'Gracias' : 'Thanks'}}</p>
  </div>
</div>
```

Referencias locales en plantillas

lunes, 2 de octubre de 2017 22:29

Son variables que a nivel de la plantilla hacen referencia a un elemento del DOM, sea un estándar HTML o un componente, permitiendo manipular su valor desde la propia plantilla en respuesta a determinados eventos.

```
<div>
  <h2>Usando #</h2>
  <form>
    <label for="nombre">Dime tu nombre (y pulsa enter)</label>
    <input type="text" id="nombre" name="nombre" #nombre>
    <button (click)="nombre.value = ''">Borrar</button>
  </form>
  <p>Hola {{nombre.value}}</p>
</div>
```

referencia local el elemento input

Las referencias locales pueden ser accedidas desde la vista gracias al decorador @ViewChild

```
@ViewChild("<referencia>") <variable>: ElementRef;
```

El tipo corresponde a la referencia general a cualquier elemento del DOM

Este decorador suele utilizarse en la gestión de los formularios, como veremos.

Existen los decoradores
@ViewChild / @ViewChildren
@ContentChild / @ContentChildren

Los segundos están relacionados con el acceso a elementos procedentes de la proyección de contenidos (transclusión de AngularJS)

Encapsulación de la vista

domingo, 15 de octubre de 2017 10:27

```
@Component({  
    selector:  
    templateUrl:  
    ...  
    encapsulation:  
    ...  
})
```

Metadato que define el tipo de encapsulación de la vista que se utilizará

ViewEncapsulation.Native
ViewEncapsulation.Emulated
ViewEncapsulation.None.

ViewEncapsulation.Native

Utiliza el Shadow DOM definido en el nuevo estándar de *Web Components*

- los estilos del componente son totalmente independientes
- los estilos del componente no son visibles fuera de él
- los estilos externos NO se aplican en el componente

Puede no ser reconocido por todos los navegadores

Es interesante en componentes que se van a reutilizar y que contienen todas sus reglas de estilo

ViewEncapsulation.Emulated

Emula en cierto modo el anterior aplicando CSS estándar

- es el valor por defecto
- los estilos del componente no son visibles fuera de él
- los estilos externos SI se aplican en el componente, siempre que no sean sobreescritos por estilos internos

ViewEncapsulation.None.

No hay ninguna encapsulación

Comunicación entre componentes

miércoles, 13 de septiembre de 2017 19:22

Formas de comunicación

Comunicación entre un componente padre (contenedor) y un componente hijo (incluido en el anterior)

- Configuración de propiedades (Padre → Hijo)
- Envío de eventos (Hijo → Padre)

- Invocación de métodos (Padre → Hijo)
 - Con variable *template*
 - Inyectando hijo con *@ViewChild*
- Compartiendo el mismo servicio (Padre ↔ Hijo)

Los inyectables (servicios) son objetos *singleton* y por tanto compartidos entre los distintas clases que los instancian

<https://angular.io/docs/ts/latest/cookbook/component-communication.html>

Configuración de propiedades

miércoles, 13 de septiembre de 2017 19:54

(Padre → Hijo)

El componente padre puede especificar propiedades en el componente hijo como si fuera un elemento nativo HTML

vista
(padre) → <hijo [title]='appTitle'></hijo>

El valor de title en el hijo corresponderá a la propiedad appTitle en el padre

class controller :
(hijo) → @Input()
private title: string;

vista
(hijo) → <h1>{{title}}</h1>

Envío de eventos

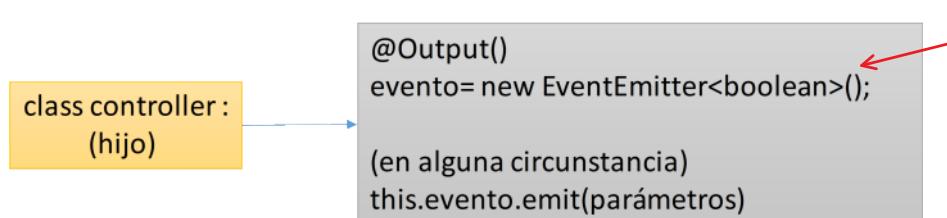
miércoles, 13 de septiembre de 2017 19:56

(Hijo → Padre)

El componente hijo puede generar eventos que son atendidos por el padre como si fuera un elemento nativo HTML

El padre se suscribe al evento : le asigna una función manejadora.
La variable \$event apunta al evento generado

vista
(padre) → <header (evento)='hiddenTitle(\$event)'></header>



No funciona si se declara y
luego se inicializa en el OnInit()

Ejemplo de comunicación entre componentes



Cuándo crear un componente

miércoles, 13 de septiembre de 2017 22:18

- Cuando la lógica y/o el *template* sean suficientemente complejos
- Cuando los componentes hijos puedan reutilizarse en varios contextos

Los ejemplos del curso (y otros similares) suelen ser demasiado sencillos para que compense la creación de componentes hijos

Directivas de Angular

domingo, 15 de octubre de 2017 10:27

son atributos específicos de angularJS que se pueden asignar a cualquier etiqueta HTML.

- El atributo o etiqueta se denomina `ng-nombre`;
- el método asociado `ngNombre`

un elemento del DOM queda "marcado" para que Angular le asigne un determinado comportamiento, que puede suponer incluso una transformación de ese elemento del DOM o alguno de sus hijos



Si modifican la estructura del DOM se denominan **directivas estructurales**.
Su nombre comienza por *

Directivas estructurales

**NgFor
*NgIf
NgSwitch

Otras directivas

*NgStyle
NgClass
NgNonBindable*



Iteraciones

*ngFor

Directiva estructural que genera el recorrido a una colección (un *array* o un objeto del modelo) indicándole la variable local a su ámbito donde se almacenará el elemento actual de cada iteración.

en la clase aElementos = [.....]

Existe un *array* en el modelo

```
<tag_html *ngFor="(let) elemento of aElementos | filtros | ordenación">  
  ... {{elemento}} ...  
</tag_html>
```

Similitud con el bucle **for ... in** de JS

Más adelante veremos las opciones de filtrado y ordenación

<https://angular.io/docs/ts/latest/api/common/index/NgFor-directive.html>

Ámbito de las iteraciones



Técnicamente, el código HTML iterado tendrá un ámbito (*scope*) local

- en él se pueden definir propiedades específicas de este ámbito, que se inicializan mediante let
- además existen una serie de propiedades ya predefinidas que toman valor dinámicamente conforme se realizan las sucesivas iteraciones y que pueden ser recogidas en propiedades del controlador

La más importante es **index** que devuelve en cada momento el índice en el recorrido del elemento actual sobre el que se está iterando

```
*ngFor="let elemento of aElementos; let i = index">
```

Scope de las iteraciones



El conjunto de las variables que almacena automáticamente el ámbito (*scope*) de una iteración es el siguiente

index	Number	Iterator offset of the repeated element (0..length-1)
first	Boolean	True, if the repeated element is first in the iterator
middle	Boolean	True, if the repeated element is between first and last in the iterator
last	Boolean	True, if the repeated element is last in the iterator
even	Boolean	True, if the iterator position \$index is even (otherwise, false)
odd	Boolean	True, if the iterator position \$index is odd (otherwise, false)

Pensamientos: ejemplo de iteraciones

Conforme añadimos ideas, creamos un *array* que mostramos, iterando sobre el, en la parte interior

Junto con la iteración, vemos de nuevo como funciona el doble *binding*.

Pensamientos

Indica tu nombre

Comparte una idea

Hola Pepe

Pensamientos que has tenido

- Tu pensamiento numero 1 fue: "Una idea"
- Tu pensamiento numero 2 fue: "Segunda idea"
- Tu pensamiento numero 3 fue: "No se me ocurre nada"
- Tu pensamiento numero 4 fue: "Ya termino"

Alejandro L. Cerezo - Madrid 2015

Condiciones: ngIf



Se puede controlar si un elemento aparece o no en la página dependiendo del valor de un atributo de la clase usando la directiva `ngIf`

- dependiendo del valor del atributo booleano `visible`

```
<p *ngIf="visible">Text</p>
```

- dependiendo de una **expresión**

```
<p *ngIf="num == 3">Num 3</p>
```

Combinando ngFor / ngIf



No se pueden incluir dos directivas estructurales (de tipo *) en el mismo elemento

```
<li *ngFor="let elem of elems" *ngIf="elem.check">  
  {{elem.desc}}  
</li>
```

La solución más simple es anidar elementos HTML (divs), cada uno con su directiva

También es posible usar la versión de las directivas sin el azúcar sintáctico (*), en su versión extendida con el elemento *template* (que no aparece en el DOM)

```
<template ngFor let-elem [ngForOf]="elems">  
  <li *ngIf="elem.check">{{elem.desc}}</li>  
</template>
```

<https://github.com/angular/angular/issues/4792>

Angular 4

miércoles, 04 de octubre de 2017 9:05

```
<div *ngIf="aldeas.length > 0; then ideasTemplate else vacioTemplate"></div>

<ng-template #ideasTemplate class="lista">
  <h2>Lista de ideas</h2>
  <ul><li *ngFor="let idea of aldeas">{{idea}}</li></ul>
</ng-template>

<ng-template #vacioTemplate>
  <p>Escribe alguna idea</p>
</ng-template>
```

Muestrario: mostrar y ocultar (1)

Uso de la directiva

***ngIf**

- asociándola al evento clic en un botón
- asociándola al paso del ratón sobre una imagen

Separamos el *footer* del fichero principal y lo incorporamos con otro componente

Muestrario

Ratones inalambricos disponibles en los siguientes colores



Ocultar Colores Disponibles

Rojo

Verde

Azul

ngSwitch / *ngSwitchCase

martes, 17 de octubre de 2017 22:37

```
<ul *ngFor="let person of aUsuarios"
    [ngSwitch]="person.country">
    <li *ngSwitchCase="'UK'"
        class="text-success">>{{ person.name }} ({{ person.country }})
    </li>
    <li *ngSwitchCase="'USA'"
        class="text-primary">>{{ person.name }} ({{ person.country }})
    </li>
    <li *ngSwitchCase="'SP'"
        class="text-danger">>{{ person.name }} ({{ person.country }})
    </li>
    <li *ngSwitchDefault
        class="text-warning">>{{ person.name }} ({{ person.country }})
    </li>
</ul>
```

ngSwitch define la variable que determinara los casos

ngSwitchCase define cada uno de los casos

Ejercicio

lunes, 16 de octubre de 2017 22:51

Entrada de datos:

- Autor →
- Título

Creación de una lista

- ngFor
- ngIf

The screenshot shows a web application with a red header containing the title "Libros!" and the "icono TRAINING CONSULTING" logo. Below the header is a white content area divided into two sections: "Datos del libro" and "Lista de Libros".

Datos del libro

Two input fields are present: "Titulo:" and "Autor:", each with a corresponding text input box. Below these fields are two buttons: "Añadir" (highlighted with a blue border) and "Borrar".

Lista de Libros

A list of books is displayed with two items:

- Neuromante (William Gibson)
- Fundación (Isaac Asimov)

At the bottom right of the content area, there is a footer with the text "Alejandro Cerezo Lasne - 2017" and "Icono Training Consulting".

Estilos iniciales



Existen varias formas de definir inicialmente un CSS en Angular 2

- **Globalmente** en el **fichero css** asociado al index.html
- Local al componente:
 - En la propiedad styles de @Component
 - En el **fichero css** definido en styleUrls de @Component
 - En el template

Gestión de estilos



Directamente

- Asociar un estilo concreto de un elemento a un atributo

Mediante clases (Buena práctica)

- Asociar la clase de un elemento a un atributo de tipo string
- Activar una clase concreta con un atributo boolean
- Asociar la clase de un elemento a un atributo de tipo objeto (mapa de string a boolean)



Style (1)

[style.<nombre>]

permite asignar a un elemento del DOM una propiedades de estilo almacenadas como un string en el modelo de la aplicación

```
<p [style.color] ="estiloColor">
```

Se pueden indicar fácilmente las unidades

```
<p [style.fontSize.em] ="pSizeEm">Text</p>
```

```
<p [style.fontSize.%] ="pSizePerc">Text</p>
```



Style(2)

[ngStyle]

permite asignar a un elemento del DOM una o varias propiedades de estilo almacenadas como un objeto en el modelo de la aplicación

```
<p [ngStyle] ="estiloColor">
```

En el controller:

```
estilos = {"bachgrouun-color" : "green",  
          "color": "silver"}
```

Como en cualquier otro contexto CSS, no es una buena práctica la aplicación directa de estilos, siendo más recomendable la aplicación de las clases adecuadas a cada caso

class (1)



[class] permite asignar a un elemento del DOM una clase mediante expresiones que hacen referencia al modelo.
De esa forma al modificar el modelo se aplican clases diferentes y se modifica el aspecto del elemento

La propiedad del modelo puede tener varios formatos:

- una **cadena de caracteres** con uno o más nombres de clases

```
<h1 [class]="nombreClase">Title!</h1>
```

class controller : → nombreClase =..."

El cambio del valor de la variable se refleja en la aplicación de diferentes clases dinámicamente

class (2)



[class.<nombre>] = "boolean"

Es posible activar una clase concreta con un atributo boolean y se puede hacer con diversas clases

```
<h1      [class.red]="redActive"
           [class.yellow]="yellowActive">
    Title!
</h1>
```

class controller : → redActive ="true"
 yellowActive ="false"

class (3)



[ngClass] permite asignar a un elemento del DOM una clase mediante expresiones más complejas

[ngClass] = "array"

Es posible aplicar diversas clases cuyos nombres se almacenan en un array

```
<h1      [ngClass]="['class1', 'class2', 'class3']"  
Title!</h1>
```

[ngClass] = "objeto"

- las claves permiten especificar nombres de clases y
- sus valores corresponden a expresiones que deben cumplirse para que éstas se apliquen.

```
<p [ngClass]="{positivo: total>=0, negativo: total<0}">
```

Ejemplo: acumulador

domingo, 24 de septiembre de 2017 15:15

The screenshot shows a presentation slide with a yellow background and a red Angular.js logo in the top-left corner. The title 'Acumulador con clases' is displayed prominently in blue text. Below the title, there are two sections: 'Ejemplos de clases que se aplican dinámicamente' and a button labeled 'Acumulador con clases'. The 'Acumulador con clases' section contains a list of bullet points:

- en respuesta a una selección por el usuario
- en función del valor de un dato del modelo

Below this list is a form titled 'Control de operación:' with an 'Incremento' input field set to '10' and two buttons ('+' and '-'). A note below the form states 'En el acumulador llevamos 10'. At the bottom right, there is copyright information: 'Alejandro L. Cerezo, CLE Formación Madrid - 2015'.



Lista de Tareas / Componentes

A screenshot of a web browser window titled "127.0.0.1:8080". The page has a header "A checklist" and a message "There are no items yet.". Below is a form with a text input "Enter the description..." and a "Add" button.

Cada tarea será
un componente

A screenshot of a web browser window titled "127.0.0.1:8080". The page has a header "A checklist" and displays three items in a list: "Leeche" (checked), "Pan" (checked), and "Galletas" (unchecked). Each item has a "Delete" button next to it. Below the list is a text input "Vino" and an "Add" button.

Refactorizamos la
aplicación de
gestión de tareas

Directivas propias

lunes, 27 de noviembre de 2017 22:53

<https://angular.io/guide/attribute-directives>

<https://www.codementor.io/christiannwamba/build-custom-directives-in-angular-2-jlqrk7dpw>

<https://www.concretepage.com/angular-2/angular-2-custom-directives-example>

```
import { Directive, ElementRef} from '@angular/core';

@Directive({
  selector: '[appPrueba]',
})
export class PruebaDirectiva {
  constructor(private eTarget: ElementRef) {}
}
```

El selector tiene el formato de atributo, para que la directiva sea utilizada como tal

el parámetro eTarget es el elemento al que se aplica la directiva que es injectado en aquella. Equivale a \$('selector') sobre el elemento. Al ser de tipo ElementRef, incluye la propiedad nativelement que corresponde al elemento en si

Una vez creada, la directiva se utiliza como cualquier otra directiva de Angular

```
<p appPrueba>Contenido del párrafo</p>
```

Atributos

Se definen como en cualquier componente, decorando una propiedad de la clase con el decorador @Input. Lo habitual es que la propiedad corresponda al propio nombre de la directiva

```
import { Directive, ElementRef} from '@angular/core';

@Directive({
  selector: '[appPrueba]',
})
export class PruebaDirectiva {

  @Input() appPrueba: string; ←

  constructor(private eTarget: ElementRef) {}
}
```

Eventos

El decorador @HostListener (<nombre de evento>) se puede aplicar a un método para convertirlo en manejador del mencionado evento

Pipes

jueves, 14 de septiembre de 2017 13:41

Filter/Pipe	Angular 1.x	Angular 2	
<i>currency</i>	✓	✓	definen el aspecto de los valores monetarios
<i>date</i>	✓	✓	definen en diversos detalles el formato de una fecha; permiten utilizar una serie de formatos ya predefinidos
<i>lowercase</i>	✓	✓	
<i>uppercase</i>	✓	✓	
<i>titlecase</i>		✓	definen el uso de mayúsculas y minúsculas
<i>json</i>	✓	✓	
<i>limitTo</i>	✓		formatea la salida de un objeto completo
<i>slice</i>		✓	
<i>number</i>	✓		presenta parte de un array
<i>decimal</i>		✓	
<i>percent</i>		✓	
<i>i18nSelect</i>		✓	definen el número de decimales de un dato
<i>i18nPlural</i>		✓	
<i>async</i>		✓	mapean los datos de un array
<i>orderBy</i>	✓		
<i>filter</i>	✓		

Formatos de fechas

<i>yyyy</i>	Four-digit representation of year (for example, AD 1 => 0001, AD 2010 => 2010)
<i>yy</i>	Two-digit representation of year, padded (00–99) (for example, AD 2001 => 01, AD 2010 => 10)
<i>y</i>	One-digit representation of year (for example, AD 1 => 1, AD 199 => 199)
<i>MMMM</i>	Month in year (January–December)
<i>MMM</i>	Month in year (Jan-Dec)
<i>MM</i>	Month in year, padded (01–12)
<i>M</i>	Month in year (1–12)
<i>dd</i>	Day in month, padded (01–31)
<i>d</i>	Day in month (1–31)
<i>EEEE</i>	Day in week (Sunday–Saturday)
<i>EEE</i>	Day in week (Sun-Sat)

<i>HH</i>	Hour in day, padded (00–23)
<i>H</i>	Hour in day (0–23)
<i>hh</i>	Hour in AM/PM, padded (01–12)
<i>h</i>	Hour in AM/PM, (1–12)
<i>mm</i>	Minute in hour, padded (00–59)
<i>m</i>	Minute in hour (0–59)
<i>ss</i>	Second in minute, padded (00–59)
<i>s</i>	Second in minute (0–59)
<i>.sss</i>	Millisecond in second, padded (000–999)
<i>o,sss</i>	
<i>a</i>	AM/PM marker
<i>Z</i>	Four-digit (+sign) representation of the time zone offset (-1200 – +1200)
<i>ww</i>	ISO 8601 week of year (00–53)
<i>w</i>	ISO 8601 week of year (0–53)

Atajos

Parámetro	Descripción	Ejemplo
<i>medium</i>	equivalent to 'MMM d, y h:mm:ss a' for en_US locale	Sep 3, 2010 12:05:08 PM
<i>short</i>	equivalent to 'M/d/yy h:mm a' for en_US locale	9/3/10 12:05PM
<i>fullDate</i>	equivalent to 'EEEE, MMMM d, y' for en_US locale	Friday, September 3, 2010
<i>longDate</i>	equivalent to 'MMMM d, y' for en_US locale	September 3, 2010
<i>mediumDate</i>	equivalent to 'MMM d, y' for en_US locale	Sep 3, 2010
<i>shortDate</i>	equivalent to 'M/d/yy' for en_US locale	9/3/10
<i>mediumTime</i>	equivalent to 'h:mm:ss a' for en_US locale	12:05:08 PM
<i>shortTime</i>	equivalent to 'h:mm a' for en_US locale	12:05 PM

```
import { LOCALE_ID, NgModule } from '@angular/core';
import { registerLocaleData } from '@angular/common';
import localeEs from '@angular/common/locales/es';

registerLocaleData(localeEs);

providers: [ { provide: LOCALE_ID, useValue: 'es' } ],
```

Pipes i18N

Pipes propios

Lunes, 27 de noviembre de 2017 22:53

Directiva Pipe
- Metadato: name

Implementa el interface
PipeTransform, que supone
crear una función transform

```
import { Pipe, PipeTransform } from '@angular/core';
@Pipe({
  name: 'prueba'
})
export class PruebaPipe implements PipeTransform {
```

```
  transform(value: any, args?: any): any {
    // código que modifica value
    return value
  }
}
```

- al menos un argumento
- devuelve el resultado de modificarlo de acuerdo con la lógica de cada filtro concreto

Pipes "puros": la función transform SOLO recibe como argumento el valor que tiene que modificar

Pipes "con atributos": la función transform recibe más argumentos, que de alguna forma determinan como se realiza la transformación

Pipe puro

```
import { Pipe, PipeTransform } from '@angular/core';
@Pipe({
  name: 'capitalizar'
})
export class CapitalizarPipe implements PipeTransform {
```

```
  transform(text: string): string | null {
    if (text != null) {
      return text.substring(0, 1).toUpperCase() +
        text.substring(1);
    }
  }
}
```

Pipes "con atributos"

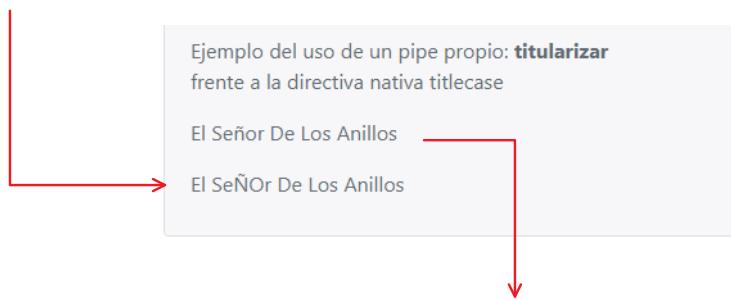
```
import { Pipe, PipeTransform } from '@angular/core';
@Pipe({
  name: 'truncar'
})
export class TruncarPipe implements PipeTransform {
  transform(value: string, limit: number = 10): any {
    return (value.length > limit) ? value.substr(0, limit) + '...' : value;
  }
}
```

Argumento secundario (limit), en este caso con un valor por defecto definido en la forma de ES6

Alternativas a Pipes nativos

domingo, 28 de enero de 2018 23:20

Existe un pipe en Angular, titleCase, cuyo funcionamiento en castellano es incorrecto



```
import { Pipe, PipeTransform } from '@angular/core';
@Pipe({
  name: 'titularizar'
})
export class TitularizarPipe implements PipeTransform {
  transform(pTexto: string): any {
    if (pTexto.length === 0) {
      return pTexto;
    }
    const aCaracteres = pTexto.split('');
    aCaracteres[0] = aCaracteres[0].toUpperCase();
    for (let i = 0; i < aCaracteres.length - 2; i++) {
      // Si (aCaracteres[i] === ' ' || aCaracteres[i] === '.' || aCaracteres[i] === ',')
      {
        aCaracteres[i + 1] = aCaracteres[i + 1].toUpperCase();
      }
    }
    return aCaracteres.join('');
  }
}
```

si no tenemos realmente un string no continuamos

mediante el método split del objeto wrapper String, obtenemos un array de caracteres correspondiente a la cadena

Mediante el uso del método toUpperCase del objeto wrapper String, podremos obtener el carácter en mayúscula del primer elemento

Analizamos el resto de la cadena, recorriendo todo el array el -2 es para evitar una excepción al salirnos del array

Si es fin de 'palabra'

Reemplazamos el siguiente elemento por su mayúscula

Finalmente, retornamos el string creado a partir del array con el método join del objeto Array

Animaciones

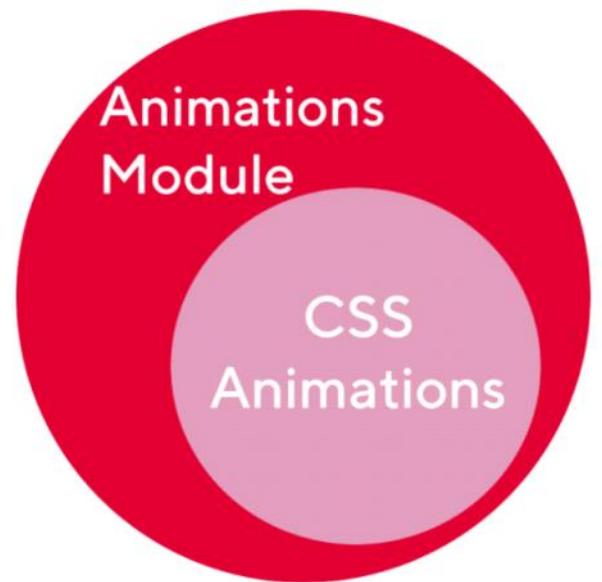
lunes, 29 de enero de 2018 23:01

Desde angular es posible generar directamente las animaciones CSS (en realidad correspondientes al estándar *Web Animations API*)

Las animaciones CSS incluyen

- transiciones
- transformaciones
- animaciones complejas (*keyframes*)

el paso de un conjunto de propiedades CSS a otro siguiendo un determinado patrón a lo largo de un intervalo de tiempo



En Angular se basan en utilizar, en el decorador del componente, el metadato `animations` con las propiedades

trigger,
state,
style,
animate,
transition,
keyframes

funciones temporizadoras
(*timing functions*)

Valor	Descripción
<code>ease</code>	<i>Default value. Specifies a transition effect with a slow start, then fast, then end slowly (equivalent to cubic-bezier(0.25,0.1,0.25,1))</i>
<code>linear</code>	<i>Specifies a transition effect with the same speed from start to end (equivalent to cubic-bezier(0,0,1,1))</i>
<code>ease-in</code>	<i>Specifies a transition effect with a slow start (equivalent to cubic-bezier(0.42,0,1,1))</i>
<code>ease-out</code>	<i>Specifies a transition effect with a slow end (equivalent to cubic-bezier(0,0,0.58,1))</i>
<code>ease-in-out</code>	<i>Specifies a transition effect with a slow start and end (equivalent to cubic-bezier(0.42,0,0.58,1))</i>
<code>step-start</code>	<i>Equivalent to steps(1, start)</i>
<code>step-end</code>	<i>Equivalent to steps(1, end)</i>
<code>steps(int,start/end)</code>	<i>Specifies a stepping function, with two parameters. The first parameter specifies the number of intervals in the function. It must be a positive integer (greater than 0). The second parameter, which is optional, is either the value "start" or "end", and specifies the point at which the change of values occur within the interval. If the second parameter is omitted, it is given the value "end"</i>
<code>cubic-bezier(n,n,n,n)</code>	<i>Define your own values in the cubic-bezier function. Possible values are numeric values from 0 to 1</i>

Procedimiento

lunes, 29 de enero de 2018 21:28

A nivel del módulo principal, registramos el módulo de Angular responsable de las animaciones en el navegador

```
import { BrowserAnimationsModule } from '@angular/platform-browser/animations';
...
imports: [
  ...
  BrowserAnimationsModule,
  ...
]
```

A nivel del componente en el que tendrá lugar la animación

```
import {
  Component, OnInit,
  trigger, state, style, transition, animate
} from '@angular/core';
```

Elementos de angular /core relacionados con la animación

Los parámetros de la animación se definen mediante metadatos del decorador del componente

```
@Component({
  ...
  animations: [
    trigger('botonState', [
      state('inactive', style({
        backgroundColor: '#eee',
        transform: 'scale(1)'
      })),
      state('active', style({
        backgroundColor: '#cf8dc',
        transform: 'scale(1.2)'
      })),
      transition('inactive=>active', animate('500ms ease-in')),
      transition('active=>inactive', animate('500ms ease-out'))
    ])
  ]
})
```

trigger: permite definir diversas animaciones y asignar a cada una de ellas un nombre que la lancara

state: define los distintos estados de la animación

style: cada estado corresponde a un conjunto de valores CSS

transition: define los pasos de un estado a otro

animate: para cada transición, determina los parametros del proceso de aplicación

En un elemento del DOM se dispara la animación

```
<button class="btn btn-primary"
[@botonState]="boton.state" (click)="toogleState()">
```

se utiliza una propiedad correspondiente al *trigger* de la animación

el valor de la propiedad corresponde a uno de los estados de la animación

en este caso hay un manejador de evento responsable de los cambios en el valor del estado

animación



```
toggleState() {  
    this.boton.state =  
        this.boton.state === 'active' ? 'inactive' : 'active';  
    this.boton.label =  
        this.boton.label === 'Activar' ? 'Desactivar' : 'Activar';  
}
```

Formularios

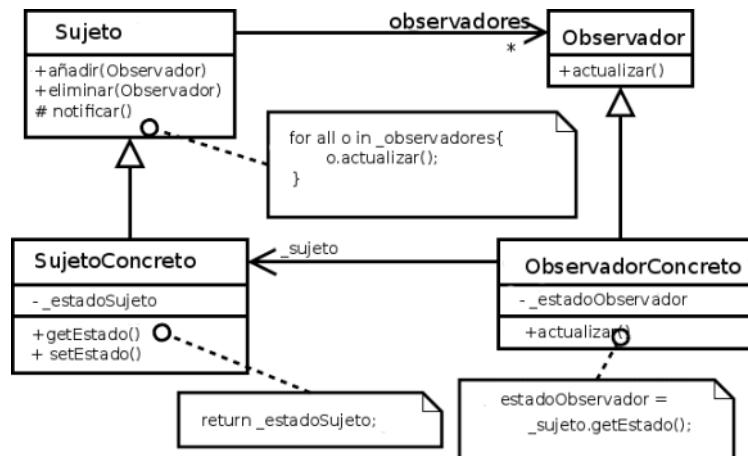
domingo, 8 de octubre de 2017 8:00

Basados en la Vista
(*Template Driven*)

- Similares a los utilizados en AngularJS
- La detección del cambio sigue el patrón *Push*
- La clave está en el doble *binding*
- Se necesita importar el módulo *FormsModule*

Basados en el Modelo
(*Model Driven*)

- Emplean programación reactiva
- La detección del cambio sigue el patrón *Observer (Pull)*
- Se necesita importar el módulo *ReactiveFormsModule*
- Son la opción recomendable en Angular



Elementos de HTML y Bootstrap

martes, 7 de noviembre de 2017 20:56

Elementos de un formulario:

- `input type text...`
- `text`
- `password`
- `email | tel | url`
- `search`
- `number | range`
- `color`
- `submit`
- `date | datetime | datetime-local`
- `month | week | time`
- `textbox`
- `input type checkbox`
- `input type radio`
- `select/options`

Formato básico en Bootstrap

Directivas y formularios

A

input
textarea
select

[(ngModel)]: indica la propiedad del modelo.
a la que se asocia el elemento

```
<label for="nif">NIF:</label>  
<input id="nif" type="text" id = "nif" name="nif"  
[(ngModel)]="oSeguro.nif" />
```

Para poder prescindir del atributo *name* dentro de un *form*, hay que modificar las propiedades del formulario

```
<label for="casado">Casado:</label>  
<input id="casado" type="checkbox" id="casado" name="casado"  
[(ngModel)]="oSeguro.casado" />
```

Se suele decir:

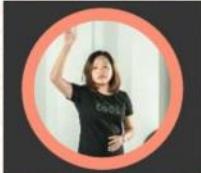
“Si el valor de una directiva ng-model no incluye un punto es que está mal.”

Formularios: más información

A

[https://scotch.io/tutorials/
how-to-deal-with-different-
form-controls-in-angular-2](https://scotch.io/tutorials/how-to-deal-with-different-form-controls-in-angular-2)

Jecelyn Yeen



Angular 2 Form Controls

Code Demo angular2 angular5 javascript typescript

How to Deal with Different Form Controls in Angular 2

How to Deal with Different Form Controls in Angular 2 (with latest forms module)

jecelyn Yeen JUL 18, 2016 Tutorials Comments 0 🔍

RadioButtons y Checkboxes



ngModel: como en cualquier control de formulario, indica la propiedad del modelo asociada al elemento

Radio-buttons → cada conjunto de ellos comparte el mismo valor de ngModel

Checkboxes → Si la propiedad tiene tipo, debe ser boolean

```
<input type="checkbox" [(ngModel)]="angular"/>
<label>Angular</label>
<input type="checkbox" [(ngModel)]="javascript"/>
<label>JavaScipt</label>
```

class controller : → angular:boolean
javascript:boolean

Checkboxes y arrays



***ngFor**: permite utilizar un array de objetos asociado a un conjunto de controles

```
<span *ngFor="let item of items">
    <input type="checkbox"
        [(ngModel)]="item.selected"/> {{item.value}}
</span>
```

```
class controller : items: Array<Object> = [
    {value:'Item1', selected:false},
    {value:'Item2',selected:false}
]
```

Checkboxes en Angular 1.x



Checkboxes

ngTrueValue: permite asignar un valor personalizado al elemento cuando el campo *checkbox* está marcado.

ngFalseValue: es lo mismo que ngTrueValue, pero en este caso con el valor asignado cuando el campo no está marcado.

ngChange: sirve para indicar operaciones a realizar cuando se produce un evento de cambio en el elemento. Se dispara cuando cambia el estado del campo, marcado a no marcado y viceversa. El valor puede ser una expresión o una llamada a una función del *scope*.

Radio-Buttons



ngModel: se asocia a una misma propiedad del controller en todos los RB de un grupo.

su valor vera el correspondiente al atributo value del RB seleccionado

```
<input type="radio" name="gender"
[(ngModel)]="genero" value="Male"><label>Male</label>
<input type="radio" name="gender"
[(ngModel)]="genero" value="Female"><label>Female</label>
```

class controller : → genero: string

Formularios: select / options



En HTML, cada "option" puede tener 2 componentes

```
<option value="valor opcional">Etiqueta</option>
```

Angular únicamente añade la directiva **ngModel** para recoger el valor (si existe) o la etiqueta de la opción seleccionada

```
<select name="country" ng-model="user.country">
  <option value="">Please select an option</option>
  <option value="US">United States</option>
  <option value="GB">United Kingdom</option>
  <option value="AU">Australia</option>
</select>
```

Angular añade una opción en blanco a no ser que exista una con valor ""

Select / options desde el modelo



***ngFor
ngValue**

permite **crear automáticamente** un select/options a partir de un conjunto de datos, procesando un array de objetos (altems)

```
<select id="select" [(ngModel)]="resultado">  
<option *ngFor="let elem of aElementos" [ngValue]="elem">  
{{elem.nombre}} </option>
```

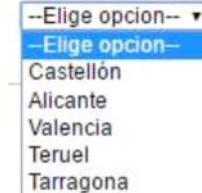
ngValue es el responsable de recoger los valores de los elementos (en este caso objetos completos) y de que el seleccionado se almacene en la variable asociada a ngModel

corresponde al ngOptions de Angular 1.x



Ejemplo de select/options

```
provincias=[  
    {idProvincia:2, nombre:"Castellón"},  
    {idProvincia:3, nombre:"Alicante"},  
    {idProvincia:1, nombre:"Valencia"},  
    {idProvincia:7, nombre:"Teruel"},  
    {idProvincia:5, nombre:"Tarragona"}  
];  
provinciaSeleccionada=null
```



```
<select [(ngModel)] = "provinciaSeleccionada"  
        <option *ngFor = "let provincia of provincias"  
               [ngValue] = "provincia"> {{provincia.nombre}}  
        </option>  
</select>
```



Formulario: Selección de opciones

Ejemplo de formulario con 2 de los mecanismos habituales de selección de opciones: *check box* y *select / options*

Formulario

Selección de opciones

- Imprimir resultado
- Tono claro

Provincia —Elige opción-- ▾

Resultado

- Opción print seleccionada: false
- Opción claro seleccionada: oscuro
- Provincia elegida:

Alejandro L. Cerezo - Madrid 2015

Lista de tareas (1): formulario

- Añadimos un formulario que permita recoger la descripción de una tarea y que incluya un botón añadir tarea. Mediante ngSubmit vinculamos el botón a una función manejadora.
- A continuación de cada item añadimos un botón que nos permita eliminarlo

Tareas

<input type="text" value="Describe una tarea"/>	<input type="button" value="Añadir"/>
Preparar curso de Angular	<input type="button" value="X"/>
Preparar curso de Web Components	<input type="button" value="X"/>

♥ CLE Formación - Curso de Angular



Lista de compras

A checklist

There are no items yet.

Enter the description... Add

Ejemplo de "formulario" para gestionar una lista de tareas

A checklist

<input checked="" type="checkbox"/> Leche	<input type="button" value="Delete"/>
<input checked="" type="checkbox"/> Pan	<input type="button" value="Delete"/>
<input type="checkbox"/> Galletas	<input type="button" value="Delete"/>

Vino Add

Métodos de arrays:
array.push()
array.indexOf()
array.splice()

Validación

miércoles, 13 de septiembre de 2017 0:16

form → la propia etiqueta HTML está ligada a la directiva Angular **ngForm** (i.e. es su selector), por lo que se instancia automáticamente el correspondiente objeto, que permitirá conocer en todo momento el estado del formulario y de cualquiera de sus controles.

Esta instancia es oculta, pero puede ser accedida en la propia **vista** mediante una **referencia local**

```
<form novalidate (ngSubmit)="enviar()" #myform= "ngForm"> ← referencia local que acceda a la instancia del formulario
```

El acceso desde el **modelo/controlador** se consigue gracias al decorador `@ViewChild`

```
@ViewChild('myform') form: any;  
...  
console.log(this.form); →  
  
▼ NgForm {_submitted: false, ngSubmit: EventEmitter, form: FormGroup} ⓘ  
  control: (...)  
  controls: (...)  
  dirty: (...)  
  disabled: (...)  
  enabled: (...)  
  errors: (...)  
  ▶ form: FormGroup {validator: null, asyncValidator: null, _onCollectionChange: f,  
    formDirective: (...)  
    invalid: (...)  
    ▶ ngSubmit: EventEmitter {_isScalar: false, observers: Array(1), closed: false, is  
      path: (...)  
      pending: (...)  
      pristine: (...)  
      statusChanges: (...)  
      submitted: (...)  
      touched: (...)  
      untouched: (...)  
      valid: (...)  
      value: (...)  
      valueChanges: (...)  
      _submitted: false  
    }  
  }  
  ▶ __proto__: ControlContainer
```

Requerimientos y estado

Los requerimientos de validación se establecen directamente con los nuevos atributos incorporados en HTML5

- **required**: valor booleano: cuando es true marca un campo como obligatorio.
- **max**: indica el número máximo de caracteres permitidos en un campo.
- **min**: indica el número mínimo de caracteres permitidos en un campo.
- **pattern**: Valida un campo frente a una expresión regular (regex).

El estado del formulario y de cada control viene definido por el valor de una serie de propiedades

- **Untouched**: When true, the control has not been interacted with the user
- **Touched**: When true, the control has been interacted with the user
- **Pristine**: The control and its underlying model has not been changed
- **Dirty**: The control and its underlying model has been changed

Estas propiedades permiten no mostrar mensajes de validación hasta que el usuario ha comenzado a llenar el formulario

- **Valid**: The inner model is valid
- **Invalid**: The inner model is not valid

Estas propiedades permiten determinar la validez de cualquier control para hacer visibles o no los correspondientes mensajes, e.g utilizando el atributo `hidden`.

Cuando se renderiza el HTML, aquellas propiedades que valgan true darán lugar a la aplicación de las correspondientes clases de CSS.

Estas propiedades son accesibles desde la referencia local del formulario

```
myform.form.controls.firstname
```

name asignado a cada uno de los controles

Pero es más sencillo crear referencias locales específicas para cada control

```
<input name="firstname" ... #firstnameState="ngModel">
```

Información al usuario

Si no se cumplen los requerimientos de validación, el navegador responderá en la forma que tenga predefinida en función de la validación HTML5. Para evitar estos mensajes se utiliza el atributo **novalidate** en la etiqueta form.

El siguiente paso es crear los mensajes específicos de cada situación y ocultarlos o mostrarlos en función del valor de las propiedades antes citadas. Para acceder a ellas se definen referencias locales (#) en cada uno de los controles

```
<form name="myform" novalidate (ngSubmit)="enviar()">
```

Anulamos la validación estándar HTML5

```
<input type="text" id="firstname" name="firstname" [(ngModel)]="user.firstname" required="true" minlength="2" #firstnameState="ngModel">
```

input con doble binding

requerimientos de validación

referencia local

```
<!--Mensajes de validación-->
<div class="error-message"
[hidden]="firstnameState.valid || firstnameState.pristine">
El nombre es obligatorio</div>
```

condiciones en las que se oculta el mensaje de error de validación

Para cada directiva de validación existe una propiedad errors que tomará un valor según las circunstancias, creándose un objeto correspondiente al primero de los errores que se esté produciendo

```
{{firstnameState.errors?.required}}
{{firstnameState.errors?.minlength}}
```

Para mostrarlos se utiliza el **operador Elvis**, para que solo se intente llamar la propiedad de la derecha si la de la izquierda no es nula

Ejemplo

domingo, 8 de octubre de 2017 8:38

The slide has a decorative background with a yellow vertical bar on the right and a textured pattern on the left.

Formularios: validación

Realizamos un formulario con:

- los campos Nombre y Apellido obligatorios y de un mínimo de 2 caracteres
- el campo Teléfono de exactamente 9 caracteres exclusivamente numéricos:
ng-pattern = "/^\\d{9}\$/"

Formulario

Datos personales

Nombre	<input type="text" value="P"/>
El nombre debe tener un mínimo de 2 caracteres	
Apellidos	<input type="text"/>
Teléfono	<input type="text"/>

Resultado

- Nombre:
- Apellido:
- Teléfono:

Alejandro L. Cerezo - Madrid 2015

Requerimientos de validación

- required
- minlength
- maxlength
- patern, e.g. ^\d{9}\$

Mensajes de error en la validación

Mensajes simples, cuando sólo existe un requerimiento de validación

```
<div [hidden]="item.valid || item.unouched"
      class="error-message">
    El item es obligatorio
</div>
```

Mensajes complejos, cuando existen varios requerimientos de validación

```
<div [hidden]="item.valid || item.unouched ">
  <div class="error-message"
       [hidden] ="!item.errors?.required">
    El item es obligatorio
  </div>
  <div class="error-message"
       [hidden] = "!
       item.errors?.minlength">
    El item debe tener un mínimo de ...
    caracteres
  </div>
</div>
```

Formularios reactivos

miércoles, 4 de octubre de 2017 6:54

- en lugar de `FormsModule`, se utiliza `ReactiveFormsModule`, también incluido en `@angular/forms`
- el desarrollo declarativo (en la vista es mínimo)
 - el atributo `[FormGroup]` en el elemento `form`
 - el atributo `FormControlName` para identificar a cada uno de los controles, en cierto modo en lugar del `[(ngModel)]`
- la gestión del formulario se traslada al controlador, donde se crea un objeto de la clase `FormGroup` para que se ocupe de ello invocándolo desde el correspondiente atributo de la vista
- Existen 2 posibilidades para instanciar ese objeto
 - Crear el objeto directamente, instanciando cada uno de sus componentes como `FormControl`
 - utiliza el método `group` del servicio `FormBuilder`, que tiene que ser injectado como cualquier otro servicio
 - este método tiene como parámetro un objeto en el que se definen cada uno los `FormControlName` de cada uno de los controles del formulario, de acuerdo con los valores asignados en la vista. Si es necesario, se puede indicar el valor inicial de los controles

```
<form [FormGroup]="formLibros" (ngSubmit)="enviarFormLibros()">
  <label for="titulo">Titulo</label>
  <input type="text" id="titulo" FormControlName="titulo">
  <label for="autor">Autor</label>
  <input type="text" id="autor" FormControlName="autor">
  <label for="editorial">Editorial</label>
  <input type="text" id="editorial" FormControlName="editorial">
  <label for="fecha">Fecha (Año)</label>
  <input type="text" id="fecha" FormControlName="fecha">
  <label></label>
  <button type="submit">Enviar</button>
</form>

import { Component, OnInit } from '@angular/core';
import { FormBuilder, FormGroup } from '@angular/forms';

@Component({
  selector: 'app-formulario',
  templateUrl: './formulario.component.html',
  styleUrls: ['./formulario.component.css']
})
export class FormularioComponent implements OnInit {

  // propiedad de tipo FormGroup (grupo de controles)
  // que se asociara a un formulario o subformulario (en casos complejos)
  formLibros: FormGroup;

  // Se inyecta FormBuilder para instanciar el FormGroup
  // correspondiente a la propiedad que se acaba de definir
  constructor(private FormBuilder: FormBuilder) { }
```

```
ngOnInit() {
    // Gracias al servicio FormBuilder, se instancia un FormGroup
    // p醙ole como par醟metro el objeto con la definici髇 del formulario
    // con los formControlNames asignados en la vista
    // forControlName="titulo"
    // forControlName="autor"
    // forControlName="editorial"
    // forControlName="fecha">
    this.formLibros = this.formBuilder.group({
        titulo: [],
        autor: [],
        editorial: [],
        fecha: ['2017']
    });
} // Fin del ngOnInit

enviarFormLibros () {}

}
```

Testing

domingo, 11 de febrero de 2018 13:48

Angular proporciona ya configurados un conjunto de entorno que permiten la ejecución de los tests:

- *Karma* para los test unitarios
- *Protractor*, para los test e2e



Angular proporciona ya configurados un conjunto de herramientas de *testing* que permiten ejecutar el código en los anteriores entornos:

- Jasmine
- Las herramientas de test del propio Angular
(*Angular Unit Testing Framework*:
`@angular/core/testing`)



Angular genera ficheros de especificaciones para test unitarios (`.spec.ts`) para todos los elementos creados mediante `ng generate`, incluyendo:

- componentes
- directivas
- pipes

Comandos e informes

domingo, 11 de febrero de 2018 13:51

Modificadores del comando ng test

`ng test --single-run`

Ejecuta la batería completa de test una sola vez mostrando el resultado en la ventana de comandos

`ng test --code-coverage`

Crea un informe de cobertura, indicando que parte del código está cubierta por las pruebas

`ng test --single-run --code-coverage`

combina los dos anteriores

Informe de cobertura

Por defecto se genera en formato HTML, y siempre en una carpeta coverage añadida al proyecto

app/		79.41%
app/dashboard-organizer/		100%
app/dashboard-sponzor/		100%
app/events-organizer/		99.56%
app/events-sponsor/		84.21%
app/services/		100%
app/sponsors-organizer/		100%
app/tasks-organizer/		100%
app/tasks-sponsor/		29.52%
app/users/		100%
app/widgets/		54.55%

```
1 1x export class Calculator {
2
3 1x   multiply(numberA: number, numberB: number): number{
4 1x     return numberA * numberB;
5   }
6
7 1x   divide(numberA: number, numberB: number): number{
8     if(numberB === 0){
9       return null;
10    }
11   return numberA / numberB;
12 }
13 1x }
14
```

Una opción para mostrar el informe es tener instalado el servidor http-server, basado en *Node*
<https://www.npmjs.com/package/http-server>

El servidor se instala globalmente mediante

```
npm install http-server -g
```

Y se ejecuta en la línea de comandos indicándole la carpeta que será raíz del servidor web

```
http-server coverage
```

Configuración

domingo, 11 de febrero de 2018 13:56

El funcionamiento por defecto de los tests en Angular depende de los ficheros

- karma.conf.js
- protractor.conf.js

Configuración de Karma

```
Frameworks           module.exports = function (config) {  
  config.set({  
    basePath: '',  
    frameworks: ['jasmine', '@angular/cli'],  
    plugins: [  
      require('karma-jasmine'),  
      require('karma-chrome-launcher'),  
      require('karma-jasmine-html-reporter'),  
      require('karma-coverage-istanbul-reporter'),  
      require('@angular/cli/plugins/karma')  
    ],  
    client:{  
      clearContext: false  
      // leave Jasmine Spec Runner output visible in browser  
    },  
    coverageIstanbulReporter: {  
      reports: [ 'html', 'lcovonly' ],  
      fixWebpackSourcePaths: true  
    },  
    angularCli: {  
      environment: 'dev'  
    },  
    reporters: ['progress', 'kjhtml'],  
    port: 9876,  
    colors: true,  
    logLevel: config.LOG_INFO,  
    autoWatch: true,  
    browsers: ['Chrome'],  
    singleRun: false  
  });  
};  
  
Plugins incluidos por defecto  
  
Salida del informe de cobertura  
  
Salida de datos:  
- consola  
- browser
```

Plugins

karma-mocha-reporter

salida de datos por consola en Karma con el formato habitual de Mocha

<https://www.npmjs.com/package/karma-mocha-reporter>

Focus

domingo, 11 de febrero de 2018 14:51

El modificador `f (focus)` puede usarse dentro de un fichero de especificaciones (`.spec.ts`) para definir el set de pruebas o incluso la prueba concreta que serán lo único ejecutado por karma cuando se lance mediante el comando `ng test`



Solo se ejecutará este set de pruebas

```
fdescribe('Test for XComponent', () => {...});
```

```
fit("should make something", ()=>{...});
```



Solo se ejecutará esta prueba

Test de componentes

domingo, 11 de febrero de 2018 17:21

El entorno de ejecución refleja las complejidades del entorno real del componente

En este entorno es frecuente distribuir las tareas entre distintos componente, e.g. con el modelo controlador / presentador

En un componente "controlador", es necesario declarar los componentes "presentadores" como parte de su entorno de pruebas

```
describe('TareasComponent', () => {
  let component: TareasComponent;
  let fixture: ComponentFixture<TareasComponent>;

  beforeEach(async(() => {
    TestBed.configureTestingModule({
      declarations: [
        TareasComponent,
        ItemComponent,
        ListaComponent,
        PipesComponent
      ],
      imports: [
        SharedModule,
        FormsModule
      ]
    })
    .compileComponents();
  }));
});
```

Se declara cualquier componente consumido por el *template* del que se está probando

Se importa cualquier módulo que incluya funcionalidades usadas por el componente, nativas de angular o propias (e.g. directivas o pipes)

En un componente "presentador" es necesario inicializar las propiedades decoradas como inputs cuando no son tipos elementales

```
beforeEach(() => {
  fixture = TestBed.createComponent(ListaComponent);
  component = fixture.componentInstance;
  component.aItems = [];
  fixture.detectChanges();
});
```

Se declara e inicializa una propiedad input, que normalmente recibiría su valor desde el componente "controlador"

Test de directivas

domingo, 11 de febrero de 2018 17:51

Importaciones habituales en la mayoría de los tests

```
import { Component, DebugElement, ElementRef } from '@angular/core';
import { By } from '@angular/platform-browser';
import { TestBed, ComponentFixture } from '@angular/core/testing';
```

En el caso de las directivas se puede declarar en el propio test un componente "virtual", sobre el que se aplicará la directiva

```
@Component({
    template: `<p id="test" destacar></p>`
})
class TestComponent {
```

```
describe('DestacarDirective', () => {
  let component: TestComponent;
  let fixture: ComponentFixture<TestC...
  let elem: DebugElement;

  beforeEach(() => {
    TestBed.configureTestingModule({
      declarations: [
        TestComponent,
        DestacarDirective
      ]
    }).compileComponents;
  });

  beforeEach(() => {
    fixture = TestBed.createComponent();
    component = fixture.componentInstance;
    fixture.detectChanges();
    elem = fixture.debugElement.query...
  });

  it('should create an instance', () => {
    const directive = new DestacarD...
    expect(directive).toBeTruthy();
  });
});
```

Importación de la directiva y
del componente recién
creado para probarla

El componente se trata igual que si fuera el objeto de las pruebas, instanciándolo dotado de su fixture

Una variable de tipo DebugElement representa al elemento del DOM correspondiente al componente

- Al instanciar la Directiva es necesario pasarle al constructor el elemento del DOM sobre el que la directiva se aplicará

Test de pipes

domingo, 11 de febrero de 2018 18:07

No requieren apenas configuración para su versión inicial

```
describe('TitularizePipe', () => {  
  it('create an instance', () => {  
    const pipe = new TitularizePipe();  
    expect(pipe).toBeTruthy();  
  });  
});
```

La clase correspondiente al pipe se instancia directamente para realizar las pruebas