

ANGULAR 2



ONE FRAMEWORK.
MOBILE & DESKTOP.

por Jorge Arévalo - @jorgeas80

AGENDA

1. Introducción a Angular 2
2. Arquitectura
3. Eventos de entrada de usuario
4. Formularios y validaciones
5. Inyección de dependencias
6. Sintáxis de plantillas
7. Comunicaciones
8. Tests

INTRODUCCIÓN



INTRODUCCIÓN

LIBRERÍAS Y HERRAMIENTAS



node v5.x.x & npm v3.x.x

INTRODUCCIÓN

¿LENGUAJE?



INTRODUCCIÓN

LENGUAJE



ES2015 + tipado estático

Ejercicio: taller de TypeScript (ejer01)

INTRODUCCIÓN

LIBRERÍAS Y HERRAMIENTAS



webpack
MODULE BUNDLER

La herramienta de construcción a partir de la versión estable

INTRODUCCIÓN

SCAFOLDING



YEOMAN

Muchos generadores no oficiales. Falta de estandarización

INTRODUCCIÓN

SCAFOLDING



ng1, ng2, react: [FountainJS](#)

INTRODUCCIÓN

SCAFOLDING



Herramienta oficial del equipo de Angular: cli.angular.io

INTRODUCCIÓN

¿EDITOR/IDE?



Creados con el desarrollo web en mente

GRATUITOS

INTRODUCCIÓN

EDITOR/IDE



Webstorm: el mejor IDE actual para frontend web

Propietario: ~100€/año

INTRODUCCIÓN

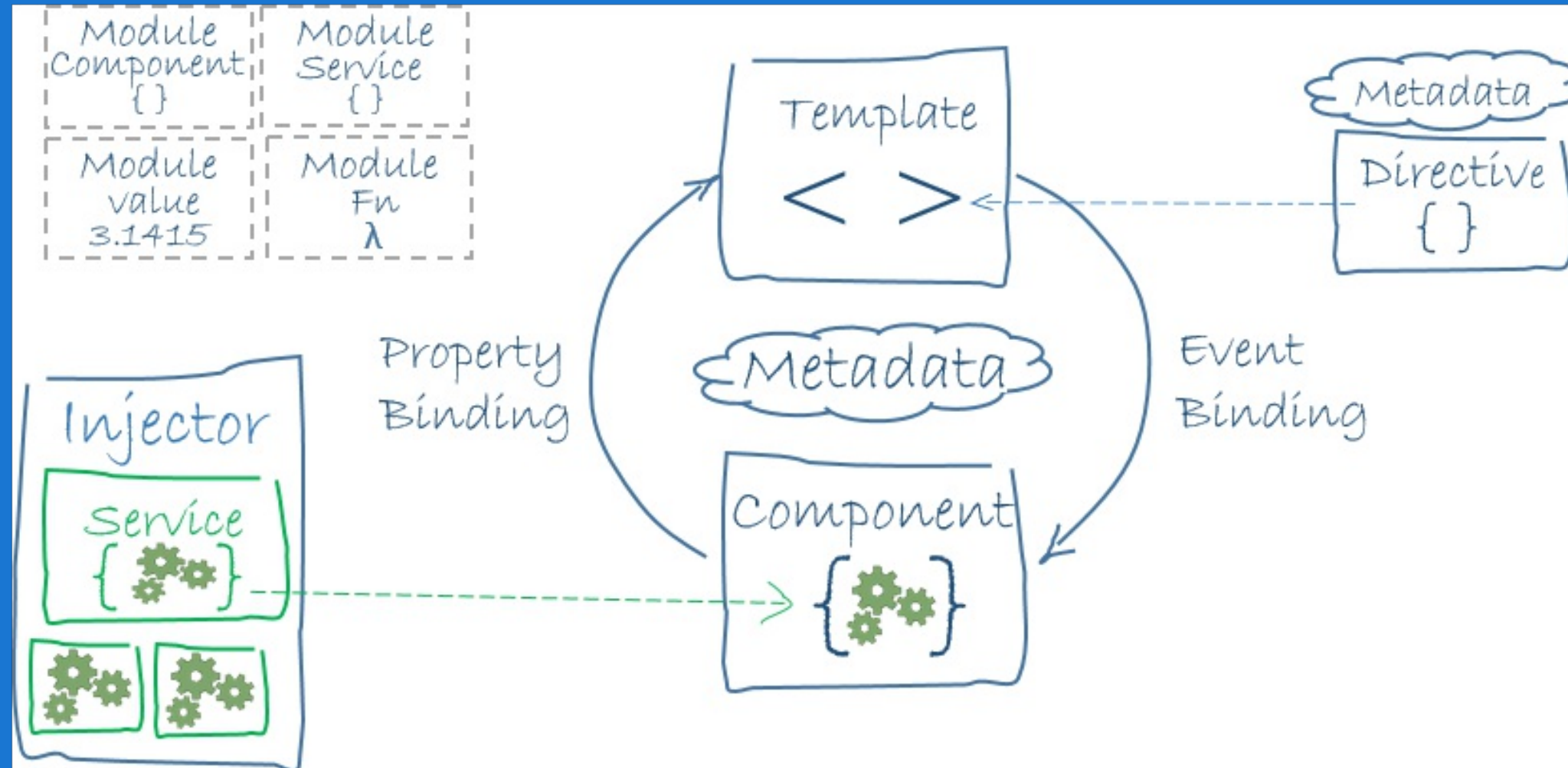
ANGULAR 2: ¿QUÉ ES?

Angular is a framework for building client applications in HTML and either JavaScript or a language (like Dart or TypeScript) that compiles to JavaScript.

— *Documentación oficial angular.io*

INTRODUCCIÓN

ANGULAR 2: BIG PICTURE



INTRODUCCIÓN

ANGULAR 2: RECURSOS

- Documentación oficial: angular.io
- Colección *curada* de recursos: [awesome angular2](#)

INTRODUCCIÓN

Ejercicio: Hola mundo en Angular 2 (ejer02)

Se puede usar

- `angular-cli` (le faltan rutas)
- QuickStart oficial con SystemJS
- FountainJS
- Hola mundo con Webpack
- Hola mundo con SystemJS
- `npm init` y a mano

ARQUITECTURA ANGULAR 2



ARQUITECTURA

Las 7 partes fundamentales de una aplicación Angular 2

- Módulos
- Components
- Plantillas
- Servicios
- Inyección de dependencias
- Directivas
- Data binding

ARQUITECTURA: MÓDULOS

Las aplicaciones Angular 2 son modulares

Angular 2 tiene su propio sistema de módulos: **NgModules**

Toda aplicación Angular 2 tiene un módulo principal,
normalmente llamado **AppModule**

No confundir con los módulos JavaScript introducidos en
ES6

ARQUITECTURA: MÓDULOS JAVASCRIPT

En JavaScript, cada fichero es un **módulo**, y todos los objetos definidos dentro del mismo, pertenecen a ese módulo

Los objetos que queramos declarar públicos dentro del módulo, se preceden por la palabra *export*

Otros módulos (ficheros JavaScript) pueden acceder a los objetos exportados mediante sentencias *import*

ARQUITECTURA: MÓDULOS

JAVASCRIPT

```
//----- lib.js -----  
export const sqrt = Math.sqrt;  
export function square(x) {  
  return x * x;  
}  
export function diag(x, y) {  
  return sqrt(square(x) + square(y));  
}
```

```
//----- main.js -----  
import { square, diag } from 'lib';  
console.log(square(11)); // 121  
console.log(diag(4, 3)); // 5
```

ARQUITECTURA: MÓDULOS

ANGULAR 2

Un módulo de Angular no es otra cosa que una **clase** con un **decorador**

Un decorador es una función que modifica una clase JavaScript.

El decorador que se usa para declarar una clase como módulo es **@NgModule**

ARQUITECTURA: MÓDULOS

ANGULAR 2

Ejemplo de módulo Angular 2

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { AppComponent } from './app.component';
@NgModule({
  imports: [BrowserModule],
  declarations: [AppComponent],
  bootstrap: [ AppComponent ]
})
export class AppModule { }
```

Angular 2 se compone de varios módulos que importaremos según los necesitemos.

ARQUITECTURA: MÓDULOS

ANGULAR 2

Los metadatos usados por *NgModule* tienen este significado

- **declarations** las vistas del módulo (componentes, directivas o filtros)
- **exports** declaraciones usables por otros módulos
- **imports** objetos exportados por otros módulos para poder ser usados en éste
- **providers** creadores de servicios
- **bootstrap** la aplicación principal. Solo el módulo principal debería usar esta propiedad.

ARQUITECTURA: MÓDULOS ANGULAR 2

Arrancar la aplicación

```
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';  
import { AppModule } from './app.module';  
  
platformBrowserDynamic().bootstrapModule(AppModule)  
  .then(success => console.log(`Bootstrap success`))  
  .catch(error => console.log(error));
```

De momento, no hace nada.

ARQUITECTURA: COMPONENTES

Dentro de un módulo habrá, entre otras cosas,
componentes

*Son nuevas etiquetas HTML con una vista
y una lógica definidas por el desarrollador*

— Micael Gallego

ARQUITECTURA: COMPONENTES

Un componente, internamente, es solo una clase

```
export class AppComponent implements OnInit {  
  ngOnInit() {  
    console.log("Component started");  
  }  
}
```

Esto NO es un componente. Es la clase que proporcionará la funcionalidad al mismo

El método *ngOnInit* es un **lifecycle hook**

ARQUITECTURA: COMPONENTES

```
@Component({
  selector: 'my-app',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent implements OnInit {
  ngOnInit() {
    console.log("Component started");
  }
}
```

Esto es un componente

Hemos añadido **metadatos** a la clase mediante un **decorador**

ARQUITECTURA: COMPONENTES

¿Qué metadatos pueden usarse para configurar el componente?

- **selector** etiqueta HTML donde se insertará el código HTML del componente
- **templateUrl** ruta, relativa al módulo actual, donde se buscará el código HTML del componente
- **stylesUrl** array con rutas, relativas al módulo actual, de fichero de hoja de estilo que el módulo podrá usar
- **providers** array de proveedores de servicios que el inyector pondrá a disposición del componente

ARQUITECTURA: PLANTILLAS

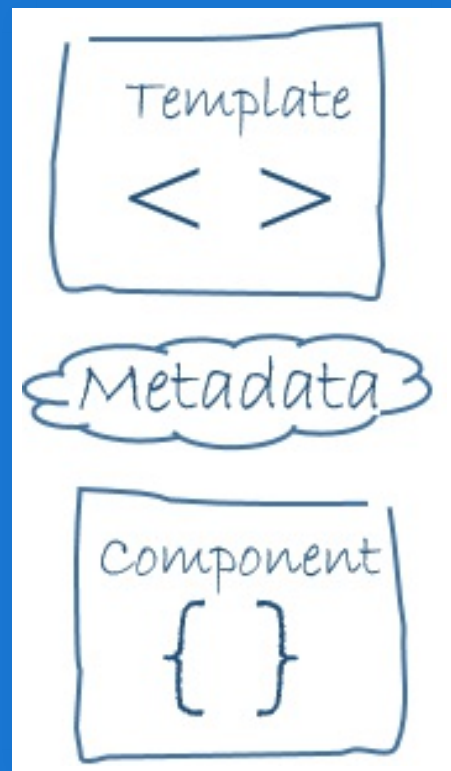
Las plantillas son ficheros con código HTML que dicen cómo renderizar el componente

Pueden incluir un cierto grado de lógica (repetición de elementos, visualización condicional...) además de estilos propios

```
<h1>Hello from Angular 2 App with Webpack</h1>  
<h2>This is the beginning of Fibonacci sequence</h2>  
<span *ngFor="let f of fib"> {{f}} </span>
```

ARQUITECTURA: COMPONENTES

Por tanto, un componente:



- Tiene una plantilla definida en un fichero HTML (la *vista* que se renderiza)
- Tiene una lógica en forma de clase TypeScript.
- Ambas partes se vinculan mediante un decorador que configura la clase pasándole metadatos

ARQUITECTURA: ESTILOS

¿Cómo añadimos estilos a nuestras plantillas?

- **Globalmente**
 - Añadiendo directamente el link en el index
 - Si usamos *angular-cli*, podemos añadir estilos a *src/styles.css* o *angular-cli.json*
- **Localmente en cada componente**
 - En la propiedad *styles* o *styleUrls* de *@Component*
 - En la propia plantilla, como estilos *inline*

Además, hay varias maneras de estilar elementos de manera condicional. Lo veremos en el tema de sintáxis de plantillas

ARQUITECTURA: ESTILOS

Ejercicio: ngIf, ngFor, data binding, estilos (ejer03)

ARQUITECTURA: SERVICIOS

Cualquier apartado de nuestra aplicación que no se ocupe de gestionar la interfaz de usuario, debería ser implementado como servicio.

Un servicio se implementa como una clase normal y corriente

```
export class Logger {  
  log(msg: any) { console.log(msg); }  
  error(msg: any) { console.error(msg); }  
  warn(msg: any) { console.warn(msg); }  
}
```

ARQUITECTURA: SERVICIOS

Como clase, puede tener un constructor que recibe parámetros, y miembros privados

```
export class HeroService {  
  private heroes: Hero[] = [];  
  
  constructor(  
    private backend: BackendService,  
    private logger: Logger) { }  
  
  getHeroes() {  
    this.backend.getAll(Hero).then( (heroes: Hero[]) => {  
      this.logger.log(` Fetched ${heroes.length} heroes.`);  
      this.heroes.push(...heroes); // fill cache (spread syntax https://goo.gl/inzydZ)  
    });  
    return this.heroes;  
  }  
}
```

ARQUITECTURA: SERVICIOS

¿Te sorprende ver un modificador de estilo en un parámetro del constructor?

```
constructor(private service: HeroService) { }
```

Es una simplificación de TypeScript

```
class Person {  
  private firstName: string;  
  private lastName: string;  
  
  constructor(firstName: string, lastName: string) {  
    this.firstName = firstName;  
    this.lastName = lastName;  
  }  
}
```

```
class Person {  
  constructor(private firstName: string, private lastName: string) {  
  }  
}
```

Compruébalo

ARQUITECTURA: SERVICIOS

¿Cómo se crea un servicio en Angular 2?

1. Se crea una nueva **clase** para el servicio
2. Se anota esa clase con **@Injectable** (lo veremos con detenimiento más adelante)
3. Se añade esa clase a la **lista de providers del NgModule** (aunque también se puede añadir a nivel de componente)
4. Se añade como **parámetro en el constructor** del componente que usa el servicio

ARQUITECTURA: INYECCIÓN DE DEPENDENCIAS

Es un **patrón de diseño**

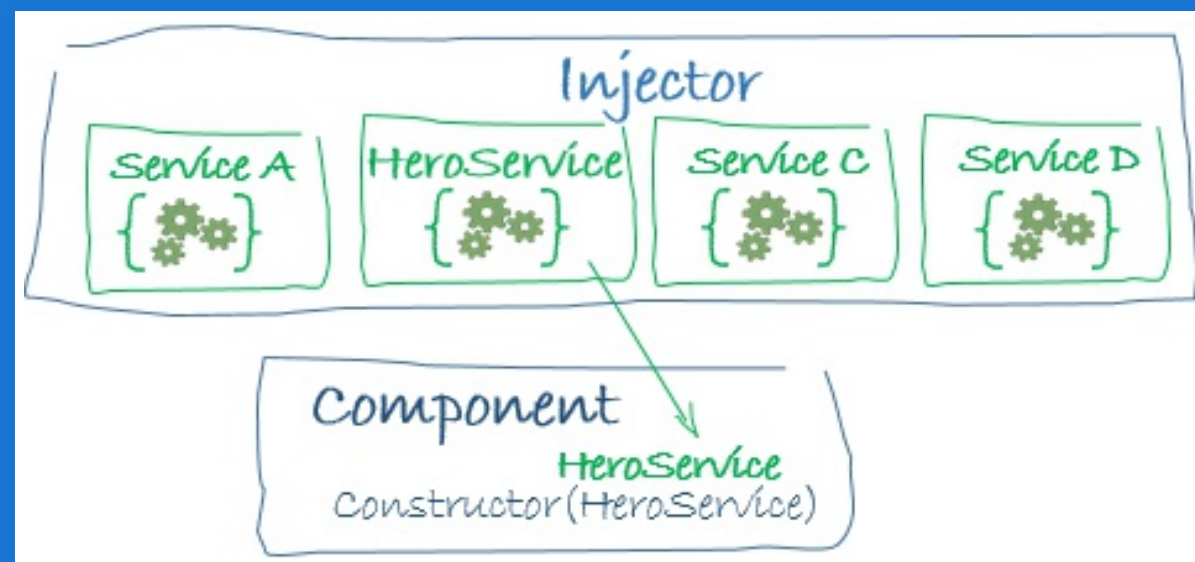
Angular 2 lo usa para proporcionar funcionalidad a componentes, en forma de instancias de clases que implementen dicha funcionalidad.

Cuando Angular 2 crea un componente, primero pregunta al **inyector** por los servicios que el componente requiere.

ARQUITECTURA: INYECCIÓN DE DEPENDENCIAS

El inyector mantiene un contenedor de servicios creados previamente, para proporcionárselos a los componentes

Si el servicio no ha sido creado aun, lo crea a partir de un **provider**, que es un molde para crear servicios.



ARQUITECTURA: INYECCIÓN DE DEPENDENCIAS

Los servicios son **Singleton** compartidos entre todos los componentes del módulo

No obstante, se pueden definir a nivel de componente

```
import { Component } from '@angular/core';
import { BooksService } from './books.service';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  providers: ['BooksService']
})
export class AppComponent {
  constructor(private booksService: BooksService){}
  /* .... */
}
```

OJO: Por cada componente que se creara, se instanciaría un servicio nuevo

ARQUITECTURA

Ejercicio (ejer04)

Modificar el ejercicio anterior para obtener los datos a partir de un servicio (probar a definirlo a nivel de módulo y de componente)

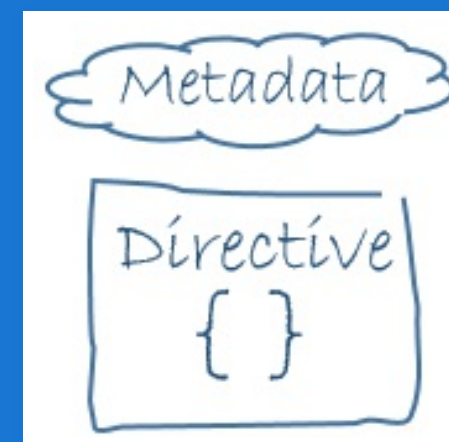
ARQUITECTURA: DIRECTIVAS

Componentes, servicios y plantillas son los tres bloques básicos de Angular 2

Pero un componente, realmente es un caso especial de *directiva*

Las directivas le dicen a Angular cómo renderizar el DOM

En el fondo, son simplemente **clases de TypeScript con metadatos**



ARQUITECTURA: DIRECTIVAS

El componente es el caso más especial de directiva, porque:

- Una directiva define un comportamiento para un elemento existente del DOM
- Un componente, además de definir comportamiento, define su propia vista (*inventa* un elemento nuevo del DOM)

Al igual que los componentes se crean con `@Component`, las directivas se crean con `@Directive`

ARQUITECTURA: DIRECTIVAS

Más en profundidad, distinguimos 3 tipos de directivas

- **@Component** realmente, el decorador @Component es un decorador @Directive extendido
- **Estructural** añaden o eliminan elementos del DOM de la página
- **Atributo** modifican la apariencia o comportamiento o apariencia de algún elemento HTML existente

ARQUITECTURA: DIRECTIVAS

Ejemplos de directivas de tipo estructural

```
<span *ngfor="let f of fib"> {{f}} </span>  
<p *ngif="showText">Show this text</p>
```

Ejemplo de directiva de tipo atributo

```
<div [ngclass]="{selected: isSelected}"> </div>
```

ARQUITECTURA: DIRECTIVAS

La sintáxis * de las directivas es solo azúcar sintáctico.

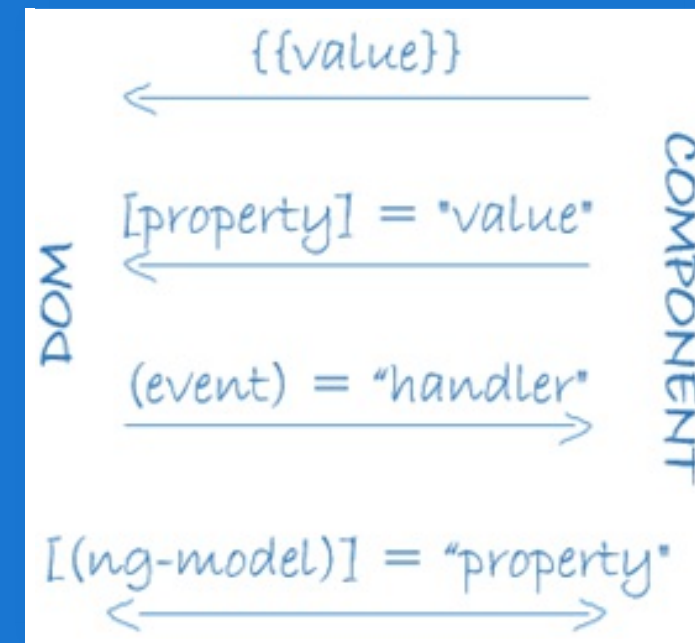
En caso de que queramos incluir dos directivas estructurales en el mismo elemento HTML, deberemos usar la sintáxis extendida

```
<template ngfor="" let-elem="" [ngforof]="elems">  
  <li *ngif="elem.check">{{elem.desc}}</li>  
</template>
```

ARQUITECTURA: DATA-BINDING

Directivas/Componentes, servicios y plantillas no son nada sin el *data binding*

Mediante un marcado especial en las plantillas, distinguimos **cuatro tipos** diferentes de data-binding.



ARQUITECTURA: DATA-BINDING

Aquí vemos tres de los cuatro tipos

```
<li>{{hero.name}}</li>  
<img [src]="imgUrl"/>  
<li (click)="selectHero(hero)"></li>
```

Y aquí el cuarto

```
<input [(ngModel)]="hero.name">
```


ARQUITECTURA: DATA BINDING

Algo importante a entender sobre el *data binding* es que los atributos HTML y las propiedades del DOM no son lo mismo

EL OBJETIVO DEL *DATA BINDING* SIEMPRE ES UNA PROPIEDAD O UN EVENTO, NO UN ATRIBUTO.

(*Objetivo* es lo que hay a la izquierda del =. Lo que hay a la derecha es la *fente*)

Vemos a continuación un cuadro resumen con ejemplos en función de cuál es el objetivo del *data-binding*

ARQUITECTURA: DATA BINDING

Binding targets / (type, target, example)

Property	Element property Component property Directive property	<pre> <hero-detail [hero]="currentHero"></hero-detail> <div [ngClass] = "{selected: isSelected}"></div></pre>
Event	Element event Component event Directive event	<pre><button (click) = "onSave()">Save</button> <hero-detail (deleteRequest)="deleteHero()"></hero-detail> <div (myClick)="clicked=\$event">click me</div></pre>
Two-way	Event and property	<pre><input [(ngModel)]="heroName"></pre>

ARQUITECTURA: DATA BINDING

Binding targets II (type, target, example)

Attribute	Attribute (the exception)	<pre><button [attr.aria-label]="help">help</button></pre>
Class	<code>class</code> property	<pre><div [class.special]="isSpecial">Special</div></pre>
Style	<code>style</code> property	<pre><button [style.color] = "isSpecial ? 'red' : 'green'"></pre>

ARQUITECTURA: DATA BINDING

De los posibles objetivos tabulados, vamos a deternos en 3

```
<!-- Objetivo es una propiedad de un componente -->  
<hero-detail [hero]="currentHero"></hero-detail>  
  
<!-- Objetivo es un evento lanzado en un componente -->  
<hero-detail (deleterequest)="deleteHero()"></hero-detail>  
  
<!-- Objetivo es un evento lanzado en una directiva no estándar -->  
<div (myclick)="clicked=$event">click me</div>
```

En los 3 casos se repite un patrón: El objetivo del data binding no es una propiedad/evento estándar del DOM de HTML5

Son propiedades y eventos que han sido creadas a través de la API de Angular 2

ARQUITECTURA: @INPUT Y @OUTPUT

Para que una propiedad o evento creado con Angular 2 pueda ser utilizado como objetivo del data binding, ha de marcarse como *@Input* o como *@Output*

Los objetivos, Angular 2 los divide en dos categorías

- **inputs** propiedades del componente destinados a recibir datos en el *data binding*
- **outputs** propiedades del componente que pueden exponer eventos (a los que otros componentes se pueden suscribir)

SINTÁXIS DE PLANTILLAS: @INPUT Y @OUTPUT

Ambos grupos se pueden especificar como atributos del decorador *@Component*

```
@Component({  
  inputs: ['hero'],  
  outputs: ['deleteRequest'],  
})
```

O a nivel individual

```
@Input() hero: Hero;  
@Output() deleteRequest = new EventEmitter<Hero>();
```



The diagram shows a snippet of HTML template code: `<hero-detail [hero]="currentHero" (deleteRequest)="deleteHero($event)">`. Above the code, there are two handwritten labels with arrows pointing to specific parts of the code: 'Input' with an arrow pointing to the `[hero]` attribute, and 'Output' with an arrow pointing to the `(deleteRequest)` attribute.

```
<hero-detail [hero]="currentHero" (deleteRequest)="deleteHero($event)">
```

SINTÁXIS DE PLANTILLAS: @INPUT Y @OUTPUT

El objeto *EventEmitter* se utiliza para emitir un evento

En el ejemplo anterior:

```
// Asi declaro el objeto
@Output() deleteRequest = new EventEmitter<Hero>();

// Asi lanzo el evento. El que lo recoja, recibirá "Borrado" en el argumento $event del capturador
deleteRequest.emit("Borrado");
```

En el receptor del evento, *\$event* tendrá como valor el argumento de la función *emit*

En los eventos nativos del DOM, *\$event* es siempre una instancia del *DOM Element Object*

SINTÁXIS DE PLANTILLAS: @INPUT Y @OUTPUT

Podemos utilizar un nombre público para nombrar un objetivo, diferente al nombre con el que haya sido creado

A nivel individual

```
@Output('myClick') clicks = new EventEmitter<string>();
```

O al declarar el elemento

```
@Directive({  
  outputs: ['clicks:myClick'] // propertyName:alias  
})
```

Ahora ya podríamos hacer

```
<div (myClick)="clickMessage=$event">click with myClick</div>
```


SINTÁXIS DE PLANTILLAS: @INPUT Y @OUTPUT

Como regla mnemotécnica

SI EL OBJETIVO DE UN *DATA-BINDING* (A LA IZQUIERDA DEL
=) NO ES NATIVO DEL DOM (EJ: SRC, CLASS, STYLE, ETC)
SINO QUE LO HE CREADO YO, LO TENGO QUE MARCAR
COMO *@INPUT* O COMO *@OUTPUT*

ARQUITECTURA

Ejercicio (ejer05)

Modificar el ejercicio anterior para añadir varios componentes y comunicación entre ellos mediante eventos

ENTRADA DE USUARIO



ENTRADA DE USUARIO

Las acciones del usuario causan eventos del DOM. Con Angular, podemos capturarlos y actuar.

Por ejemplo, podemos capturar un click en un botón, como ya vimos en el apartado de *data-binding*

```
<button (click)="onClickMe()">Click me!</button>
```

El nombre del evento va entre paréntesis, y la acción se especifica con *onClickMe*, una *template statement* (la veremos más detenidamente en el tema sobre plantillas)

ENTRADA DE USUARIO

El código completo del componente, con la plantilla HTML embebida

```
@Component({
  selector: 'click-me',
  template: `
    <button (click)="onClickMe()">Click me!</button>
    {{clickMessage}}`
})
export class ClickMeComponent {
  clickMessage = "";

  onClickMe() {
    this.clickMessage = 'You are my hero!';
  }
}
```

ENTRADA DE USUARIO

Podemos obtener la entrada de usuario también a partir del objeto *\$event*

```
@Component({
  selector: 'click-me2',
  template: `
    <button (click)="onClickMe2($event)">No! .. Click me!</button>
    {{clickMessage}}`
})
export class ClickMe2Component {
  clickMessage = "";
  clicks = 1;

  onClickMe2(event: any) {
    let evtMsg = event ? ' Event target is ' + event.target.tagName : "";
    this.clickMessage = (`Click #${this.clicks++}. ${evtMsg}`);
  }
}
```

ENTRADA DE USUARIO

Mediante el uso de las *template reference variables*, podemos acceder al contenido de otros campos

Definimos una variable de ese tipo en nuestro elemento HTML precediendo un identificador con el caracter *hash* (#)

```
@Component({
  selector: 'key-up2',
  template: `
    <input #box="" (keyup)="onKey(box.value)">
    <p>{{values}}</p>
  `,
})
export class KeyUpComponent_v2 {
  values = '';
  onKey(value: string) {
    this.values += value + ' | ';
  }
}
```

ENTRADA DE USUARIO

En cuanto al evento *keyup*, podemos filtrarlo con *key.enter*, sin necesidad de comprobar *\$event.keyCode*

```
@Component({
  selector: 'key-up3',
  template: `
    <input #box="" (keyup.enter)="values=box.value">
    <p>{{values}}</p>
  `,
})
export class KeyUpComponent_v3 {
  values = '';
}
```

En el ejemplo, Angular 2 filtrará todas las pulsaciones de tecla hasta que pulsemos ENTER

ENTRADA DE USUARIO

Por supuesto, hay una manera mejor de obtener en nuestro componente un valor introducido por el usuario

Utilizando `ngModel`, como veremos en el siguiente capítulo (formularios)

Ejercicio (ejer06)

FORMULARIOS



FORMULARIOS

Angular 2 incluye soporte para

- Two way data binding
- Seguimiento de cambios en valores
- Validación
- Gestión de errores

FORMULARIOS

Un flujo de trabajo habitual con formularios en Angular 2 es

1. Crear clase TS que represente el modelo de datos (opcional, si el formulario representa un objeto del modelo)
2. Crear el componente que controla el formulario
3. Crear una plantilla con el HTML del formulario
4. Enlazar el componente con los campos del formulario mediante **ngModel** y Two-way data binding
5. Crear validaciones
6. Añadir CSS que proporcione feedback visual de las acciones del usuario
7. Gestionar el envío del formulario con **ngSubmit**

FORMULARIOS: CREAR MODELO

Si queremos que el formulario directamente *alimente* a nuestro modelo de datos, creamos la clase que representará dicho modelo

```
export class User {  
  
  constructor(  
    public firstName: string,  
    public lastName: string,  
    public gender: string,  
    public hiking?: string,  
    public running?: string,  
    public swimming?: string  
  ) {}  
}
```

FORMULARIOS: CREAR COMPONENTE

Creamos el componente

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

@Component({
  selector: 'simple-form',
  templateUrl: './simpleform.component.html',
  styleUrls: ['./simpleform.component.css']
})
export class SimpleformComponent implements OnInit {
  user: User;

  constructor() { }

  ngOnInit() {
    this.user = new User("", "", "");
  }

  submitForm(form: any): void{
    console.log('Form Data: ');
    console.log(form);

    // user.firstName = form.firstName
    // ...
  }
}
```

FORMULARIOS: CREAR PLANTILLA

Una plantilla básica para formulario, usando Bootstrap

```
<div class="jumbotron">
  <h2>Template Driven Form</h2>
  <form #form="ngForm" (ngSubmit)="submitForm(form.value)">
    <div class="form-group">
      <label>First Name:</label>
      <input type="text" class="form-control" placeholder="John" #firstName="ngModel" name="firstName" [(ngModel)]="user.firstName" required>
      <div [hidden]="firstName.valid || firstName.pristine" class="alert alert-danger">First name is required</div>
    </div>
    <div class="form-group">
      <label>Last Name</label>
      <input type="text" class="form-control" placeholder="Doe" #lastName="ngModel" name="lastName" [(ngModel)]="user.lastName" required>
      <div [hidden]="lastName.valid || lastName.pristine" class="alert alert-danger">Last name is required</div>
    </div>
    <!-- CONTINUA ... -->
  </form>
</div>
```

FORMULARIOS: DATA BINDING

Esto une los datos del formulario al modelo

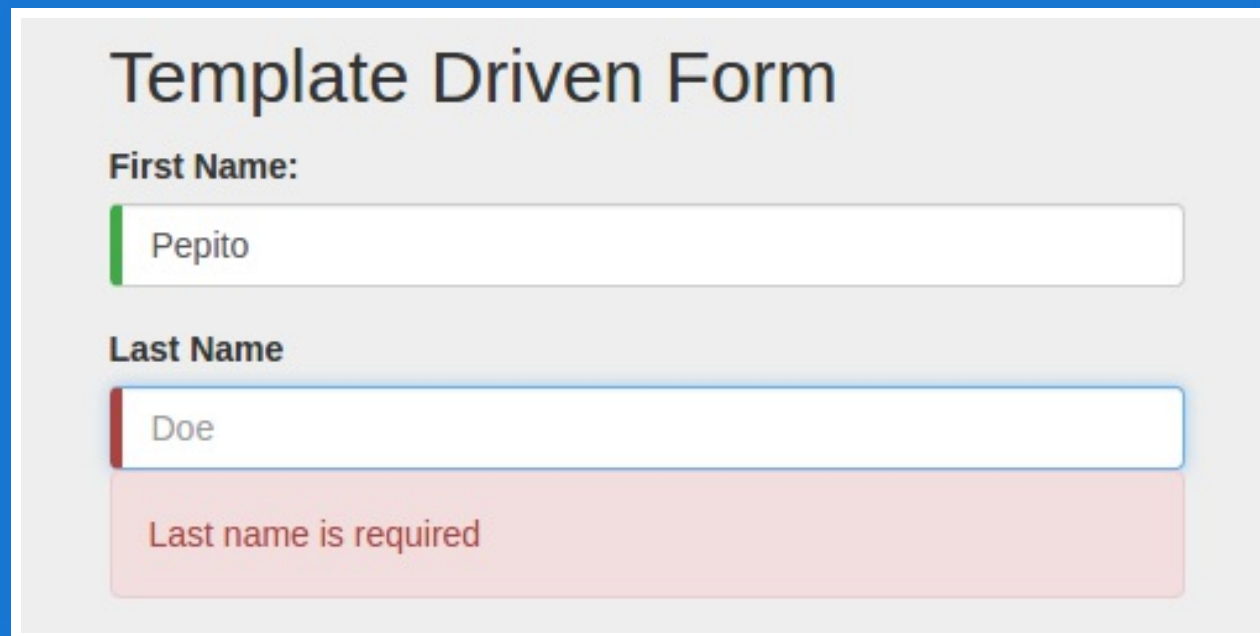
```
<input type="text" class="form-control" placeholder="John" #firstName="ngModel" name="firstName" [(ngModel)]= "user.firstName" required>
```

name **es necesario** para poder usar *ngModel*.

También necesitaremos añadir *FormsModule* al array *imports* del módulo.

FORMULARIOS: VALIDACIONES

Queremos usar este código de colores para nuestros campos



Template Driven Form

First Name:

Pepito

Last Name

Doe

Last name is required

Nos valdremos para ello de

- Las clases *ngValid/ngInvalid* de Angular 2
- El atributo *required* de HTML5

FORMULARIOS: FEEDBACK VISUAL

Creamos la siguiente hoja de estilos

```
.ng-valid[required], .ng-valid.required {  
  border-left: 5px solid #42A948; /* green */  
}  
  
.ng-invalid:not(form) {  
  border-left: 5px solid #a94442; /* red */  
}
```

Podemos asignársela al componente mediante el uso del atributo *styles* del decorador *@Component*

FORMULARIOS: FEEDBACK VISUAL

Para mostrar mensajes de error personalizados, tenemos que acceder al valor del campo del modelo a validar desde la propia plantilla (validamos campos del modelo, no valores del formulario)

Dicho de otro modo, necesitamos **acceder al valor de la directiva ngModel de un campo**

FORMULARIOS: FEEDBACK VISUAL

Usamos una *template reference variable* para acceder al controlador Angular de otro input del formulario, enlazado con el modelo mediante *ngModel*

```
<input type="text" class="form-control" placeholder="John" #firstName="ngModel" name="firstName" [(ngModel)]="user.firstName" required>
```

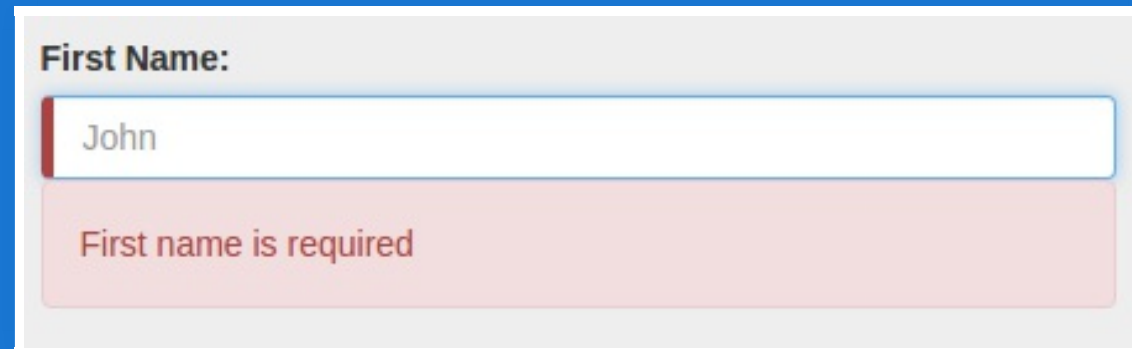
Asignamos a *#firstName* el valor de *ngModel* porque es el valor con el que se exporta la directiva *ngModel*

Si nos limitáramos a poner *#firstName* sin más, como hicimos en el tema de entrada de usuario, a lo que podríamos acceder desde la plantilla es al valor del **HTMLInputElement** con el que la API del navegador envuelve a un elemento *input* común

FORMULARIOS: FEEDBACK VISUAL

Ahora podríamos mostrar u ocultar un campo de aviso usando la validación

```
<div [hidden]="firstName.valid || firstName.pristine" class="alert alert-danger">First name is required</div>
```



First Name:

First name is required

FORMULARIOS: ENVÍO DE FORMULARIO

Antes de enviar el formulario, podemos querer hacer comprobaciones.

```
<form #form="ngForm" (ngSubmit)="submitForm(form.value)">  
  <!-- Formulario aqui -->  
</form>
```

También hemos usado otra *template reference variable*, en concreto *#form*

La directiva *ngForm* envuelve al **HTMLFormElement** que representa el formulario

Es una directiva que Angular 2 instancia y guarda información sobre las validaciones

FORMULARIOS: ENVÍO DE FORMULARIO

Ahora podemos activar o desactivar el botón de envío en función de la validez del formulario

```
<button type="submit" [disabled]="form.invalid" class="btn btn-default">Submit</button>
```

FORMULARIOS

Angular 2 provee otras maneras de validar formularios. Por ejemplo, *Reactive Forms*

Ejercicio: Formulario con validaciones en Angular 2
(ejer07)

INYECCIÓN DE DEPENDENCIAS



INYECCIÓN DE DEPENDENCIAS

It's a coding pattern in which a class receives its dependencies from external sources rather than creating them itself.

— *angular.io docs*

INYECCIÓN DE DEPENDENCIAS

La intención Transferir la responsabilidad de instanciar las dependencias de una clase al usuario de la misma.

La solución (sub-óptima) Crear una **factoría** para construir las dependencias y la clase

```
import { Engine, Tires, Car } from './car';
// BAD pattern!
export class CarFactory {
  createCar() {
    let car = new Car(this.createEngine(), this.createTires());
    car.description = 'Factory';
    return car;
  }
  createEngine() {
    return new Engine();
  }
  createTires() {
    return new Tires();
  }
}
```

INYECCIÓN DE DEPENDENCIAS

Otra solución mejor Delegar la responsabilidad de crear la clase a partir de sus dependencias en un **inyector**

Al inyector le pasamos tanto la clase a construir como sus dependencias y se encarga del trabajo

```
injector = ReflectiveInjector.resolveAndCreate([Car, Engine, Tires]);  
let car = injector.get(Car);
```

INYECCIÓN DE DEPENDENCIAS

Angular 2 incluye su propio inyector de dependencias. No hace falta que lo creamos explícitamente.

Solo tenemos que asegurarnos de dos cosas

- De que los elementos que queremos que sean inyectados son decorados con *@Injectable*
- De que definimos el array *providers* de elementos a inyectar a nivel de módulo o de componente

INYECCIÓN DE DEPENDENCIAS

Si queremos servicios que estén disponibles para todos los componentes del módulo, definimos el array a nivel de módulo

Si queremos servicios solo para un componente, definimos el array a nivel de ese componente

Más información en las [FAQ de NgModule](#)

INYECCIÓN DE DEPENDENCIAS

Un servicio inyectable puede, a su vez, requerir de otros servicios.

```
import { Injectable } from '@angular/core';
import { HEROES } from './mock-heroes';
import { Logger } from '../logger.service';
@Injectable()
export class HeroService {
  constructor(private logger: Logger) { }
  getHeroes() {
    this.logger.log('Getting heroes ...');
    return HEROES;
  }
}
```

INYECCIÓN DE DEPENDENCIAS: PROVIDERS

El array de *providers* suele tener este aspecto

```
providers: [Logger]
```

Pero realmente, eso es *azúcar sintáctico* para la forma extendida (token + receta)

```
providers: [{ provide: Logger, useClass: Logger }]
```

A provider is a recipe for delivering a service associated with a token.

— *angular.io docs*

INYECCIÓN DE DEPENDENCIAS: PROVIDERS

El primer elemento del objeto *provider* es el **token**: la clase que representa la dependencia.

El segundo elemento es el *provider definition object*: La vía para instanciar la clase en cuestión.

La versión simplificada es, como ya hemos visto, equivalente a instanciarla clase token usando la propia clase en si. Pero hay más maneras.

INYECCIÓN DE DEPENDENCIAS: PROVIDERS

Puede darse la situación en la que queramos sustituir una clase por otra mejorada que siga la misma interfaz.

Para eso usamos *useExisting*

```
{ provide: OldLogger, useExisting: NewLogger}
```

En el ejemplo, *OldLogger* es un **alias** para *NewLogger*

INYECCIÓN DE DEPENDENCIAS: PROVIDERS

Un caso de uso avanzado de esta técnica es reducir la **complejidad de la API** ofrecida por un servicio.

```
{ provide: MinimalLogger, useExisting: LoggerService }
```

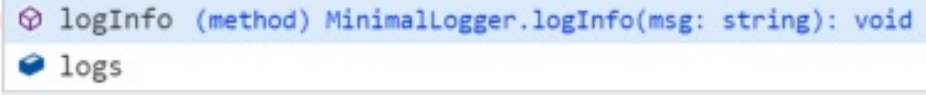
Imaginemos que *LoggerService* es un servicio muy complejo, con 20 métodos. Pero a nosotros solo nos interesa 1. Es el que vamos a añadir a *MinimalLogger*

```
// class used as a restricting interface (hides other public members)
export abstract class MinimalLogger {
  logInfo: (msg: string) => void;
  logs: string[];
}
```

INYECCIÓN DE DEPENDENCIAS: PROVIDERS

Si ahora accediéramos al objeto provisto por esa dependencia, solo veríamos los campos de *MinimalLogger*

```
this.logs = logger.logs;  
logger.logInfo('sta
```



MinimalLogger es una **class-interface**

A nivel abstracto, se comporta como una interfaz. Pero ha de implementarse como una clase, porque **las interfaces no son objetos JavaScript**. Solo existen en TypeScript. Se pierden al transpilar a JavaScript.

INYECCIÓN DE DEPENDENCIAS: PROVIDERS

Otra situación posible es proporcionar como dependencia un objeto ya instanciado, en lugar de hacer que el inyector lo instancie.

Usamos *setValue* para ello

```
// An object in the shape of the logger service
let silentLogger = {
  logs: ['Silent logger says "Shhhhh!". Provided via "useValue"'],
  log: () => {}
};
```

```
[{ provide: Logger, useValue: silentLogger }]
```

INYECCIÓN DE DEPENDENCIAS: PROVIDERS

Otra posible situación es que la dependencia a instanciar tenga a su vez dependencias solo disponibles en tiempo de ejecución

```
export class LogDebugger {  
  constructor(private enabled: boolean) {}  
  
  debug(message) {  
    if (this.enabled) {  
      console.log("DEBUG: ${message}");  
    }  
  }  
};
```

INYECCIÓN DE DEPENDENCIAS: PROVIDERS

Para ello utilizamos *useFactory*, que nos permite crear una función que instancie la dependencia pasándole parámetros

```
[{  
  provide: LogDebugger,  
  useFactory: () => {  
    return new LogDebugger(true);  
  }  
}]
```

INYECCIÓN DE DEPENDENCIAS: PROVIDERS

¿Y si el servicio a instanciar tiene dependencia de otros servicios?

```
// console.service.ts
export class ConsoleService {
  log(message) {
    console.log(message);
  }
}
```

```
//log-debugger.service.ts
import { ConsoleService } from './console.service';
@Injectable()
export class LogDebugger {
  constructor(private consoleService: ConsoleService, private enabled: boolean) {}

  debug(message) {
    if (this.enabled) {
      this.consoleService.log('DEBUG: ${message}');
    }
  }
}
```


INYECCIÓN DE DEPENDENCIAS: PROVIDERS

Podemos seguir usando *useFactory*, y pasarle un array de dependencias como tercer argumento

```
providers: [  
  ConsoleService,  
  {  
    provide: LogDebugger,  
    useFactory: () => {  
      return new LogDebugger(consoleService, true);  
    },  
    deps: [ConsoleService]  
  }  
]
```

INYECCIÓN DE DEPENDENCIAS: NON-CLASS DEPENDENCIES

¿Y si lo que queremos inyectar no es una clase, sino una cadena, un número, un array o una función?

Ya hemos visto que, mediante *useValue*, podemos inyectar un objeto instanciado directamente

Pero en el caso más sencillo, el token que nos dice cómo se ha instanciado el objeto es **una clase**

¿Y si no tenemos una clase capaz de modelar el valor a instanciar?

INYECCIÓN DE DEPENDENCIAS: NON-CLASS DEPENDENCIES

Veámoslo con un ejemplo

```
export interface AppConfig {  
  apiEndpoint: string;  
  title: string;  
}  
  
export const HERO_DI_CONFIG: AppConfig = {  
  apiEndpoint: 'api.heroes.com',  
  title: 'Dependency Injection'  
};
```

Si queremos pasar *HERO_DI_CONFIG*, ¿qué usamos como token? No tenemos una clase *AppConfig*

```
// FAIL! Can't use interface as provider token  
[  
  { provide: AppConfig, useValue: HERO_DI_CONFIG }  
]
```

```
// FAIL! Can't inject using the interface as the parameter type  
constructor(private config: AppConfig){ }
```

INYECCIÓN DE DEPENDENCIAS: NON-CLASS DEPENDENCIES

Usamos un Opaque Token

```
import { OpaqueToken } from '@angular/core';  
  
export let APP_CONFIG = new OpaqueToken('app.config');
```

Ya podemos pasarlo en el array *providers*

```
providers: [{ provide: APP_CONFIG, useValue: HERO_DI_CONFIG }]
```

O mediante el uso de *@Inject*, inyectarlo donde sea necesario

```
constructor(@Inject(APP_CONFIG) config: AppConfig) {  
  this.title = config.title;  
}
```

INYECCIÓN DE DEPENDENCIAS: @INJECT E @INJECTABLE

¿Confundido con *@Injectable* e *@Inject*?

Supongamos que tenemos un componente que hace uso de un servicio

```
@Component({
  selector: 'my-app',
  template: `
    <ul>
      <li *ngfor="let item of items">{{item.name}}</li>
    </ul>
  `,
  providers: [DataService]
})
class AppComponent {
  items:Array<any>;
  constructor(dataService: DataService) {
    this.items = dataService.getItems();
  }
}</any>
```

INYECCIÓN DE DEPENDENCIAS: @INJECT E @INJECTABLE

El servicio es éste

```
class DataService {  
  items:Array<any>;  
  
  constructor() {  
    this.items = [  
      { name: 'Christoph Burgdorf' },  
      { name: 'Pascal Precht' },  
      { name: 'thoughttram' }  
    ];  
  }  
  
  getItems() {  
    return this.items;  
  }  
}
```

Sin problema hasta aquí

INYECCIÓN DE DEPENDENCIAS: @INJECT E @INJECTABLE

Ahora supongamos que nuestro servicio usa otro servicio a su vez

```
import { Http } from '@angular/http';

class DataService {
  items:Array<any>;

  constructor(http:Http) {
    ...
  }
  ...
}
```

Se producirá un error

Cannot resolve all parameters for DataService(?). Make sure they all have valid type or annotations.

INYECCIÓN DE DEPENDENCIAS: @INJECT E @INJECTABLE

El problema es la ausencia de metadatos en el código de
DataService

El inyector de dependencias de Angular 2 funciona gracias a los metadatos generados mediante el uso de **decoradores**

Por ejemplo, un componente sabe de dónde sacar sus dependencias gracias al decorador *@Component*

En nuestro caso, *DataService* carece de metadatos, así que no sabe de dónde sacar las dependencias.

INYECCIÓN DE DEPENDENCIAS: @INJECT E @INJECTABLE

Una posible solución: usar *@Inject*

```
import { Inject } from '@angular/core';
import { Http } from '@angular/http';

class DataService {
  items:Array<any>;

  constructor(@Inject(Http) http:Http) {
    ...
  }
  ...
}
```

@Inject se encarga de *preguntar* por los metadatos necesarios para resolver una dependencia

INYECCIÓN DE DEPENDENCIAS: @INJECT E @INJECTABLE

Otra solución más elegante: Decorar los servicios que dependan de otros con *@Injectable*

```
import { Injectable } from '@angular/core';
import { Http } from '@angular/http';

@Injectable()
class DataService {
  items:Array<any>;

  constructor(http:Http) {
    ...
  }
  ...
}</any>
```

INYECCIÓN DE DEPENDENCIAS: `@INJECT` E `@INJECTABLE`

@Injectable solo sirve para forzar la generación de metadatos en el elemento decorado

De hecho, su uso se considera una buena práctica

Decora todos tus servicios con *@Injectable* y así no tendrás que preocuparte de si requerirán inyección explícita o no.

INYECCIÓN DE DEPENDENCIAS

Ejercicio: Modificar ejer04 (ejer08)

Modificar el ejercicio ejer04 para:

- Probar el uso de useFactory/useClass
- Probar a pasarle un valor constante

SINTÁXIS DE PLANTILLAS



SINTÁXIS DE PLANTILLAS

Básicamente, Angular 2 acepta todo HTML válido en una plantilla a excepción de la etiqueta `<script>`, que la ignora.

Por supuesto, se admiten componentes y directivas creadas por Angular 2, por librerías o por nosotros mismos.

Los elementos del DOM serán actualizados dinámicamente gracias a alguna de las formas de *data binding* existentes

SINTÁXIS DE PLANTILLAS: INTERPOLACIÓN

El modo más sencillo de *data binding* es la interpolación

```
<h3>
  {{title}}
  
</h3>
```

Lo que hay entre las llaves es una *template expression*. Algo que Angular 2 evalúa y convierte en cadena de texto

```
<!-- "The sum of 1 + 1 is 2" -->
<p>The sum of 1 + 1 is {{1 + 1}}</p>
  <!-- "The sum of 1 + 1 is not 4". Valid if getVal is a method of the host component -->
  <p>The sum of 1 + 1 is not {{1 + 1 + getVal()}}</p>
```

SINTÁXIS DE PLANTILLAS: TEMPLATE EXPRESSIONS

Lo que se puede usar en una *template expression* tiene varias limitaciones. No se pueden usar:

- Asignaciones (=, +=, -=)
- El operador *new*
- ; o , para encadenar expresiones
- Operadores de incremento o decremento: ++, --
- Operadores a nivel de bit: |, &
- Nada que esté fuera del contexto del componente donde nos encontremos (ej: no valen window, document, etc)

SINTÁXIS DE PLANTILLAS: TEMPLATE EXPRESSIONS

Sin embargo, se pueden utilizar unos operadores nuevos:

template expression operators

EL OPERADOR PIPE (|)

Se usa como filtro

```
<!-- Pipe chaining: convert title to uppercase, then to lowercase -->
<div>
  Title through a pipe chain:
  {{title | uppercase | lowercase}}
</div>

<!-- pipe with configuration argument => "February 25, 1970" -->
<div>Birthdate: {{currentHero?.birthdate | date:'longDate'}}</div>

<!-- Object to JSON -->
<div>{{currentHero | json}}</div>
```

SINTÁXIS DE PLANTILLAS: TEMPLATE EXPRESSIONS SAFE NAVIGATION OPERATOR (?)

Nos ahorra un *ngIf* o estrategia parecida para mostrar condicionalmente un atributo de un objeto

```
The current hero's name is {{currentHero?.firstName}}  
<!-- No hero, no problem! -->  
The null hero's name is {{nullHero?.firstName}}
```

SINTÁXIS DE PLANTILLAS: TEMPLATE STATEMENTS

Responden a eventos de usuario, como ya vimos en un tema anterior

```
(click)="save()"
```

Los valores prohibidos son los mismos que para las
template expressions

La excepción son los operadores de asignación (=, +=, -=) y
los de concatenación (;. ,)

Se permiten porque una *template statement*, por definición,
produce un efecto lateral (es parte del data binding)

SINTÁXIS DE PLANTILLAS: BINDING

Como ya hemos visto, hay diferentes tipos de *data-binding*
en Angular 2

Vamos a ver todos los tipos a continuación

SINTÁXIS DE PLANTILLAS: PROPERTY BINDING

Lo usamos cuando queremos asignar el resultado de evaluar una *template expression* a una propiedad de un elemento, directiva o componente

El caso más típico es asignar a una de las propiedades de un elemento del DOM un valor calculado en un componente

```
<img [src]="heroImageUrl">  
<button [disabled]="isUnchanged">Cancel is disabled</button>
```

También muy útil para asignar un valor a una propiedad de un componente creado por nosotros

```
<hero-detail [hero]="currentHero"></hero-detail>
```

SINTÁXIS DE PLANTILLAS: ATTRIBUTE BINDING

Es el único caso en el que Angular 2 permite *binding* con atributos

Sintáxis: [attr.nombre_propiedad]

Muy poco común: colspan, svg, aria (no hay propiedades equivalentes a estos atributos)

```
<table border="1">
  <!-- expression calculates colspan=2 -->
  <tbody><tr><td [attr.colspan]="1 + 1">One-Two</td></tr>

  <!-- ERROR: There is no `colspan` property to set!
  <tr><td colspan="{{1 + 1}}">Three-Four</td></tr>
  -->

  <tr><td>Five</td><td>Six</td></tr>
</tbody></table>

<!-- create and set an aria attribute for assistive technology -->
<button [attr.aria-label]="actionName">{{actionName}} with Aria</button>
```

SINTÁXIS DE PLANTILLAS: CLASS BINDING

Nos permite añadir y quitar nombres de clases de cualquier elemento

Sintáxis: [class.nombre_clase] ó [class]

```
<div [class]="badCurly">Bad curly</div>  
<!-- toggle the "special" class on/off with a property -->  
<div [class.special]="isSpecial">The class binding is special</div>
```

Para cambiar varios nombres a la vez, es mejor usar la directiva *ngClass* (lo veremos)

SINTÁXIS DE PLANTILLAS: STYLE BINDING

Nos permite modificar estilos *inline*

Sintaxis: [style.nombre_estilo]

```
<button [style.color] = "isSpecial ? 'red': 'green'">Red</button>  
<button [style.background-color]="canSave ? 'cyan': 'grey'" >Save</button>  
  
  <!-- works with unit extensions, dash-case and camelCase -->  
<button [style.font-size.em]="isSpecial ? 3 : 1" >Big</button>  
<button [style.fontSize.%"="!isSpecial ? 150 : 50" >Small</button>
```

Para cambiar varios estilos a la vez, es mejor usar la directiva *ngStyle* (lo veremos)

SINTÁXIS DE PLANTILLAS: EVENT BINDING

Nos permite reaccionar ante eventos producidos en los elementos de la página

Sintáxis: (evento)="template statement"

```
<button (click)="onSave()">Save</button>
```

Cuando se produce el evento, éste se captura en el objeto *\$event*

SINTÁXIS DE PLANTILLAS: EVENT BINDING

Si el evento se produce en un elemento HTML nativo, este objeto será una instancia del *DOM Event Object*

Si el evento ha sido creado de manera personalizada mediante *Event Emitter*, *\$event* tendrá la forma decidida por el creador del evento.

SINTÁXIS DE PLANTILLAS: NGMODEL BINDING

Nos permite exponer un elemento de nuestro modelo de datos y, al mismo tiempo, asignarle valores a través de acciones del usuario (*2-way data binding*)

```
<input [(ngmodel)]="currentHero.firstName">  
  <!-- Same effect -->  
<input [value]="currentHero.firstName" (input)="currentHero.firstName=$event.target.value">
```

Para poder usar *ngModel*, necesitamos añadir *FormsModule* en el array *imports* de nuestro módulo

```
@NgModule({  
  imports: [  
    BrowserModule,  
    FormsModule  
  ], //...  
})
```

SINTÁXIS DE PLANTILLAS: NGMODEL BINDING

El evento *ngModelChange* nos permite actuar cuando el valor capturado con *ngModel* cambie

```
<!-- If that function exists... -->  
<input [ngmodel]="currentHero.firstName" (ngmodelchange)="setUpperCaseFirstName($event)">
```

SINTÁXIS DE PLANTILLAS: DIRECTIVAS NATIVAS

Angular 2 ha reducido el número de directivas nativas con respecto a Angular 1

Veremos a continuación algunas de las más comunes

SINTÁXIS DE PLANTILLAS: NGCLASS

Alternativa a *class binding* cuando queremos cambiar varias clases a la vez

```
setClasses() {  
  let classes = {  
    saveable: this.canSave,    // true  
    modified: !this.isUnchanged, // false  
    special: this.isSpecial,   // true  
  };  
  return classes;  
}
```

```
<div [ngclass]="setClasses()">This div is saveable and special</div>
```

SINTÁXIS DE PLANTILLAS: NGSTYLE

Alternativa a *style binding* cuando queremos cambiar varios estilos a la vez

```
setStyles() {  
  let styles = {  
    // CSS property names  
    'font-style': this.canSave    ? 'italic' : 'normal', // italic  
    'font-weight': !this.isUnchanged ? 'bold'  : 'normal', // normal  
    'font-size':  this.isSpecial   ? '24px'   : '8px',    // 24px  
  };  
  return styles;  
}
```

```
<div [ngstyle]="setStyles()">  
  This div is italic, normal weight, and extra large (24px).  
</div>
```

SINTÁXIS DE PLANTILLAS: NGIF

Añade o elimina elementos del DOM en función de una expresión

```
<!-- because of the ngIf guard  
`nullHero.firstName` never has a chance to fail -->  
<div *ngIf="nullHero">Hello, {{nullHero.firstName}}</div>  
  
<!-- Hero Detail is not in the DOM because isActive is false-->  
<hero-detail *ngIf="isActive"></hero-detail>
```

OJO no es lo mismo **eliminar** del DOM (lo que hace ngIf) que **ocultar**

SINTÁXIS DE PLANTILLAS: NGIF

Ejemplos de ocultación de elementos del DOM (no se eliminan)

```
<!-- isSpecial is true -->
<div [class.hidden]="!isSpecial">Show with class</div>
<div [class.hidden]="isSpecial">Hide with class</div>

<!-- HeroDetail is in the DOM but hidden -->
<hero-detail [class.hidden]="isSpecial"></hero-detail>

<div [style.display]="isSpecial ? 'block' : 'none'">Show with style</div>
<div [style.display]="isSpecial ? 'none' : 'block'">Hide with style</div>
```

SINTÁXIS DE PLANTILLAS: NGSWITCH

Muestra un solo elemento del DOM de un conjunto de varios en función de lo que valga una expresión

```
<span [ngSwitch]="toeChoice">  
  <span *ngSwitchCase="Eenie">Eenie</span>  
  <span *ngSwitchCase="Meanie">Meanie</span>  
  <span *ngSwitchCase="Miney">Miney</span>  
  <span *ngSwitchCase="Moe">Moe</span>  
  <span *ngSwitchDefault>other</span>  
</span>
```

SINTÁXIS DE PLANTILLAS: NGFOR

Repite un bloque de elementos HTML en función de una plantilla y una lista de datos a mostrar

```
<div *ngFor="let hero of heroes">{{hero.fullName}}</div>
```

Lo que se le asigna a *ngFor* no es una *template expression*. Es una *microsyntax*

Una expresión en un pequeño lenguaje que Angular 2 interpreta. El lenguaje trae algunas cosas interesantes

SINTÁXIS DE PLANTILLAS: NGFOR INDEX

La variable *index* lleva un conteo de los elementos

```
<div *ngFor="let hero of heroes; let i=index">{{i + 1}} - {{hero.fullName}}</div>
```

Parecidas a *index* son *last*, *even* y *odd*

Más información en la [referencia de ngFor en la API de Angular 2](#)

SINTÁXIS DE PLANTILLAS: NGFOR TRACKBY

Con esta sintáxis, podemos especificar una función de seguimiento (*trackBy function*) que será aplicada a cada uno de los elementos

Esta función devolverá el valor que determina si uno de los elementos ha cambiado realmente o no

Gracias a ello, Angular 2 puede decidir si eliminar o no un elemento del DOM cuando se producen cambios en los elementos a mostrar

```
trackByHeroes(index: number, hero: Hero) { return hero.id; }
```

```
<div *ngFor="let hero of heroes; trackBy:trackByHeroes">{{hero.id}} {{hero.fullName}}</div>
```

SINTÁXIS DE PLANTILLAS: DIRECTIVAS PERSONALIZADAS

Como ya sabemos, Angular nos permite crear directivas mediante el decorador *@Directive*

Recordemos que *directiva* y *componente* no son lo mismo. La directiva añade comportamiento a un elemento del DOM ya existente. El componente crea su propio elemento del DOM

SINTÁXIS DE PLANTILLAS: DIRECTIVAS PERSONALIZADAS

A la hora de crear directivas, debemos conocer dos clases fundamentales

- **ElementRef** nos da acceso al elemento nativo del DOM.
INSEGURO.
- **HostListener** nos permite excuchar cualquier evento del host en el que se encuentre la aplicación

Angular desaconseja el uso de *ElementRef* por peligro de ataque XSS. Alguien podría insertar scripts maliciosos en nuestro DOM...

SINTÁXIS DE PLANTILLAS: TEMPLATE REFERENCE VARIABLES

Como ya hemos visto, son referencias a otros elementos del DOM

Sintáxis: *#variable* ó *ref-variable*

Permiten acceder a otros elementos del DOM (recuerda a *getElementById*)

```
<!-- phone refers to the input element; pass its `value` to an event handler -->
<input #phone="" placeholder="phone number">
<button (click)="callPhone(phone.value)">Call</button>

<!-- fax refers to the input element; pass its `value` to an event handler -->
<input ref-fax="" placeholder="fax number">
<button (click)="callFax(fax.value)">Fax</button>
```


SINTÁXIS DE PLANTILLAS: TEMPLATE REFERENCE VARIABLES

Un caso de uso interesante de TRV es acceder al controlador Angular de otro input de un formulario, como ya vimos

```
<label for="name">Name</label>
<input type="text" class="form-control" id="name"
  required
  [(ngModel)]="model.name" name="name"
  #name="ngModel" >
<div [hidden]="name.valid || name.pristine"
  class="alert alert-danger">
  Name is required
</div>
```

SINTÁXIS DE PLANTILLAS: TEMPLATE REFERENCE VARIABLES

Otro caso de uso habitual de TRV es acceder a la directiva *NgForm* que envuelve al *HTMLFormElement*, y que añade funcionalidad extra al elemento nativo

```
<form (ngSubmit)="onSubmit(theForm)" #theForm="ngForm">
  <div class="form-group">
    <label for="name">Name</label>
    <input class="form-control" name="name" required [(ngModel)]="currentHero.firstName">
  </div>
  <button type="submit" [disabled]="!theForm.form.valid">Submit</button>
</form>
```

SINTÁXIS DE PLANTILLAS

Ejercicio (ejer09)

COMUNICACIONES



COMUNICACIONES

Angular 2 se comunica con el exterior, principalmente mediante dos protocolos

- HTTP
- WebSocket

El segundo de ellos se ha popularizado con el auge de las comunicaciones en tiempo real. Pero por ahora, cubriremos solo HTTP.

COMUNICACIONES

Los navegadores modernos proporcionan 3 APIs para comunicación HTTP

- **XMLHttpRequest (XHR)**
- **JSONP** usado normalmente como *atajo* para evitar enfrentarse a la **política de mismo origen** impuesta por los navegadores para limitar las llamadas HTTP asíncronas a otros dominios
- **Fetch** Llamado a ser el reemplazo de XMLHttpRequest.
Aun **experimental**

COMUNICACIONES

La alternativa más moderna a *JSONP* para enfrentarse a la política del mismo origen es el protocolo *CORS*

CORS debe estar soportado por el servidor. Si no, tendremos que usar *JSONP* (que también debe soportarlo el servidor, pero al ser anterior, es más probable que servidores antiguos lo soporten)

COMUNICACIONES

Angular 2 simplifica las llamadas a las APIs de XHR y JSONP mediante **su propio cliente HTTP**

Ejemplo de llamada HTTP GET hecha con dicha librería

```
http.get(url).subscribe(  
  response => console.log(response.json()),  
  error => console.error(error)  
);
```

http representa una variable de tipo **Http**

COMUNICACIONES

http.get() devuelve un objeto de tipo **Observable** (veremos lo que es)

Dicho *Observable* expone un método *subscribe* que recibe dos parámetros

- La función que ejecutará cuando la petición sea **correcta**
- La función que ejecutará cuando la petición sea **errónea**

COMUNICACIONES

Para poder usar ambas apis, necesitamos importar los módulos *HttpModule* y *JSONPModule*

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { FormsModule } from '@angular/forms';
import { HttpModule, JsonpModule } from '@angular/http';

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

@NgModule({
  imports: [
    BrowserModule,
    FormsModule,
    HttpModule,
    JsonpModule
  ],
  declarations: [ AppComponent ],
  bootstrap: [ AppComponent ]
})
export class AppModule { }
```

COMUNICACIONES

Cuando trabajamos con HTTP, hemos de ser conscientes de la **naturaleza asíncrona** de las comunicaciones que hacemos

Los métodos que hacen llamadas HTTP, solo pueden devolver información cuando llegue la respuesta. Y hasta ese momento, **no se bloquean**

COMUNICACIONES

Un código así no tiene sentido

```
private service: myHttpService = ...  
let data = this.myHttpService.getData();  
console.log(data);
```

Ese código es **síncrono**. Y las comunicaciones HTTP no lo son

COMUNICACIONES

Para enfrentarse a la asincronía, JavaScript (y Angular 2 por extensión) ofrece **tres enfoques**, de más antiguo a más moderno

- **Callbacks** fácil caer en el *callback hell*
- **Promesas** el patrón *try-catch* aplicado a la programación asíncrona. Solución de facto ES2015.
- **Observables** implementación del *patrón reactor*. Solución favorita en Angular 2. Aportada por la librería **RxJS**

COMUNICACIONES

Los observables tienen, como mínimo, tres ventajas con respecto a las promesas

- Los observables son *vagos*. Se deben lanzar explícitamente. Las promesas se lanzan al crearse.
- En el caso particular de Angular 2, los observables son de tipo *cold*. Es decir No se ejecutan hasta que alguien se *suscribe* a ellos.
- Los observables se pueden cancelar. Las promesas no.

Observable = asynchronous immutable array.

COMUNICACIONES

Ejercicio (ejer10): varios ejemplos de peticiones HTTP

Ejercicio (ejer11): ejemplo de InMemoryDbService

Para el ejercicio 11 necesitaremos instalar la librería
angular-in-memory-web-api

```
npm install --save angular-in-memory-web-api
```

TESTING



TESTS

Angular 2 , al igual que su predecesor, AngularJS, fue creado con la idea de que fuera fácilmente *testable*

El equipo de Angular recomienda

- **Jasmine** para las pruebas unitarias
- **Karma** como *test runner*
- **Protractor** para los tests e2e

TESTS

Además, angular proporciona una serie de herramientas para facilitar los tests

La configuración de las diferentes herramientas de tests es bastante pesada (todo son ficheros de configuración en formato json)

Por eso, es buena idea comenzar nuestros proyectos con algun *starter project*, que ya incluye los ficheros de configuración listos para empezar a trabajar con tests integrados.

TESTS

Ya vimos anteriormente los *starter projects*, pero los oficiales más completos desde el punto de vista de los tests son:

- **Angular Quickstart** (usa SystemJS)
- **angular-cli** (usa Webpack, es más completo y está en constante evolución)

En este tema vamos a utilizar **angular-cli** para ayudarnos a implementar los tests

Sus herramientas de generación automática serán de gran ayuda a la hora de escribir tests

TESTS: COMPONENTES SIMPLES

En primer lugar, veremos como probar un componente sencillo (sin dependencias de servicios)

- Configuramos el módulo de testing de manera similar a cómo configuraríamos el módulo para su carga en la aplicación. Nos ayudará *TestBed.createComponent*
- Creamos una *ComponentFixture* a partir de nuestro componente. Es un wrapper que nos da una serie de funciones útiles.

TESTS: COMPONENTES SIMPLES

- A través de la *fixture*, podemos acceder tanto al componente original (*componentInstance*) como al elemento del DOM que encapsula el componente
- Ahora podemos cambiar variables de nuestro componente, y verificar cómo ha cambiado el DOM, gracias al método *query*, disponible a través de la *fixture*, que nos deja consultar el árbol del DOM.

TESTS: DEPENDENCIAS

Para probar un componente que tiene dependencia de un servicio, lo dicho para el componente sencillo sigue siendo válido

La diferencia es que, realmente, no llamaremos al servicio, porque puede ser muy pesado, depender de conexión de red, etc

TESTS: DEPENDENCIAS

Lo que haremos será crear un *stub* de nuestro servicio. Será un objeto con la misma forma (atributos, métodos), pero devolverá valores estáticos.

Como ya vimos en el tema de inyección de dependencias, es realmente sencillo implementar un servicio usando otro en su lugar, mediante el uso de *useClass* en el array de *providers* de un componente

Otra alternativa es usar el servicio real pero *capando* el método que queramos probar mediante un objeto de tipo *spy* de Jasmine.

TESTS: DEPENDENCIAS ASÍNCRONAS

En ocasiones, el servicio del que depende nuestro componente realizará llamadas http, o cualquier otro proceso asíncrono.

Jasmine está preparado para trabajar de manera asíncrona, pero Angular 2 nos facilita aun más la labor, mediante:

- La función **fakeAsync**
- La función **async**

TESTS: DEPENDENCIAS ASÍNCRONAS

La función **fakeAsync** crea un *sandbox* en el que se puede simular el paso del tiempo mediante llamadas a la función *tick(miliseecs)*. De esta forma, podemos programar de manera síncrona aunque hagamos llamadas asíncronas

```
describe('this test', () => {  
  it('looks async but is synchronous', <any>fakeAsync(): void => {  
    var flag = false;  
    setTimeout(() => { flag = true; }, 100);  
    expect(flag).toBe(false);  
    tick(50);  
    expect(flag).toBe(false);  
    tick(50);  
    expect(flag).toBe(true);  
  });  
});  
</any>
```

TESTS: DEPENDENCIAS ASÍNCRONAS

La función *async* crea un *sandbox* que intercepta todas las promesas lanzadas desde dentro. Solo nos tenemos que preocupar de capturar una promesa: la que devuelva la función *fixture.whenStable*

Cuando esa promesa se resuelva, estaremos seguros de que no hay llamadas asíncronas pendientes.

TESTS: DEPENDENCIAS ASÍNCRONAS

Tanto *fakeAsync* como *async* toman como argumento una función sin parámetros y devuelve una función que puede ser usado como cuerpo de un test *it* de Jasmine

Aunque *fakeAsync* es conceptualmente más sencilla, tiene algunas limitaciones. Por ejemplo: **no permite hacer llamadas XHR**

TESTS: DEPENDENCIAS ASÍNCRONAS

La cantidad de situaciones que pueden ser probadas con
Angular 2 es enorme

En el [manual avanzado oficial](#) se pueden ver muchos más
ejemplos

TESTS

Ejercicio (ejer12)

FIN DEL CURSO



SI TIENES ALGUNA DUDA, PUEDES **ENVIARME**
UN EMAIL O SEGUIRME EN TWITTER