

Material

goo.gl/5exKwP

**.\\Alumno
emerGencia2014**

✉ 0xnacho@gmail.com

 @0xnacho

Should you upgrade?



Acerca de mí

► Académico

- BSc in Computer Science
- MSc in Computer Science
- PhD Student



► Profesional

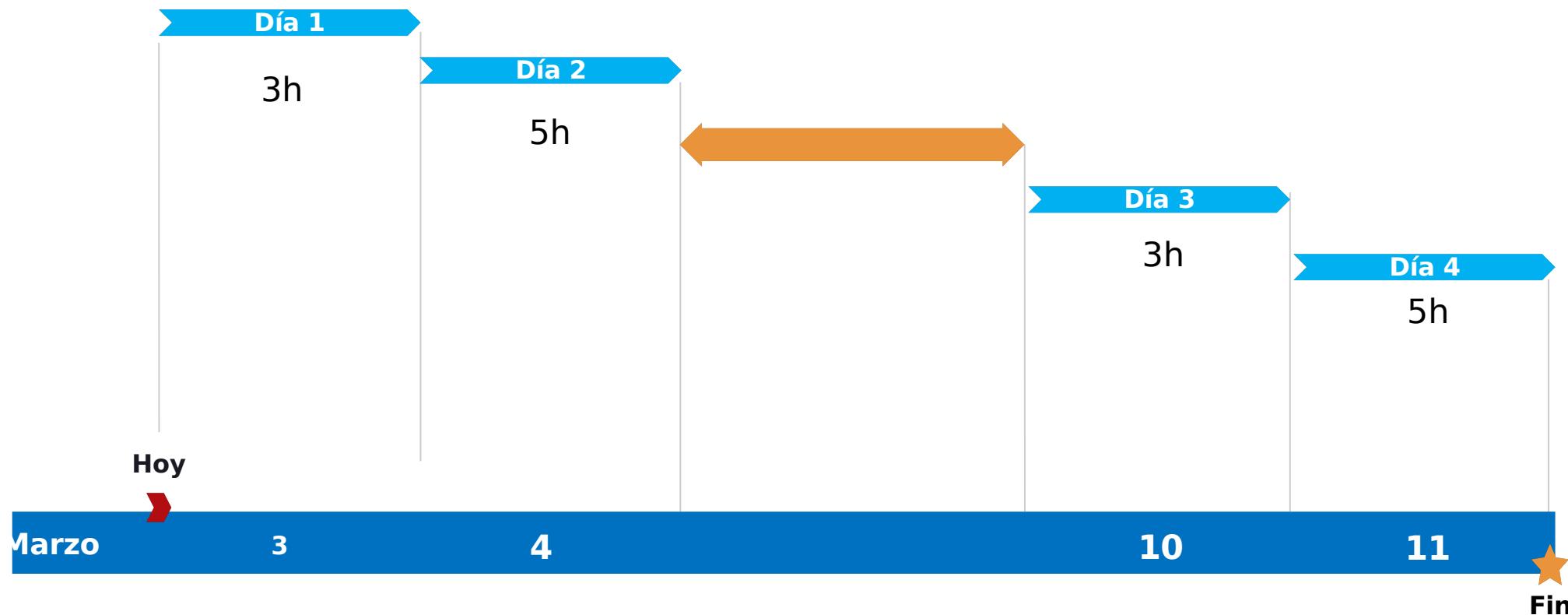
- R&D Engineer at Treelogic S.L.
- Profesor en Kschool
- Independent security researcher



Becas:
cv@treelogic.com



Agenda





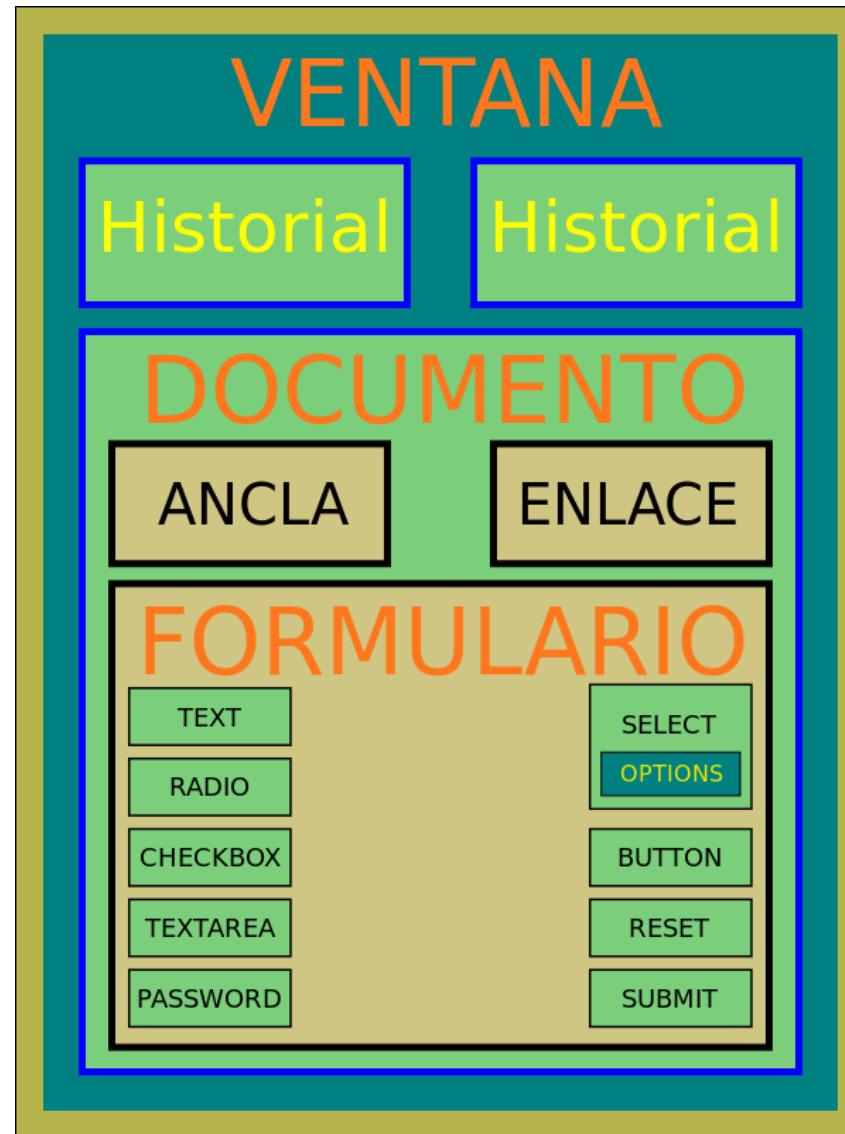
Repaso de conceptos



DOM

- ▶ **Document Object Model:** modelo de objetos del documento
- ▶ Se define como: una interfaz de plataforma que proporciona un conjunto estándar de objetos para representar documentos HTML, XHTML y XML,¹ un modelo estándar sobre cómo pueden combinarse dichos objetos, y una interfaz estándar para acceder a ellos y manipularlo
- ▶ A través del DOM los programas pueden acceder y modificar el contenido, estructura y estilo de los documentos HTML
- ▶ El responsable de mantener el DOM es el W3C (World Wide Web Consortium)
- ▶ El DOM permite ser modificado dinámicamente mediante Javascript

Jerarquía del DOM



HTML, CSS & Javascript

- ▶ **HTML**: Lenguaje de marcado utilizado para realizar páginas web
 - ▶ Mantenido por el W3C
- ▶ **CSS**: Lenguaje de marcado gráfico
 - ▶ Estiliza los documentos HTML
- ▶ **Javascript**: Lenguaje de programación interpretado
 - ▶ Utilizado en los navegadores para dinamizar las páginas HTML
 - ▶ También utilizado en el servidor (NodeJS) con fines similares a Java, Php, etc.



Javascript: el problema

- ▶ Javascript es un lenguaje utilizado en el 100% de aplicaciones web en la actualidad
- ▶ En aplicaciones grandes, es muy difícil conseguir un código fiable y estructurado
- ▶ Débilmente tipado: muy propenso a errores
- ▶ Basado en prototipos: no “podemos” definir clases
- ▶ No existe el concepto de interfaz
- ▶ No podemos modularizar nuestra aplicación: el concepto de módulo o paquete no existe
- ▶ Los IDE no nos ayudan mucho...

Javascript: el problema





Repaso de conceptos



TypeScript

- ▶ Lenguaje de programación libre y de código abierto mantenido por Microsoft
- ▶ Superconjunto de Javascript: añade tipado estático y objetos basados en clases
- ▶ Se puede utilizar para el desarrollo de aplicaciones que se ejecuten **en el lado del cliente y en el servidor** (Node.js)
- ▶ Pensado para grandes proyectos
- ▶ Angular2 está construido sobre typescript
- ▶ Útil para detectar errores en el desarrollo:
 - ▶ Uso de variables no existentes
 - ▶ Mal tipado de datos
- ▶ Proporciona funcionalidades ausentes en JS:
 - ▶ Orientación a clases
 - ▶ Interfaces
 - ▶ Tipado de variables



TypeScript : ¿cómo funciona?

1. Se escribe un programa siguiendo la especificación de TypeScript ¹.
 - ▶ TypeScript es similar a cualquier otro lenguaje orientado a objetos como Java, C#, etc.
2. Posteriormente, el lenguaje se “transpila” mediante la herramienta **tsc**
3. El lenguaje es traducido a Javascript

```
var a:number = 9;
a += 4;

function mostrar(b:string) :void{
    console.log(b);
}
mostrar('hola');
```

test.ts

tsc test.ts



```
var a = 9;
a += 4;
function mostrar(b) {
    console.log(b);
}
mostrar('hola');
```

test.js

¹ <https://github.com/Microsoft/TypeScript/blob/master/doc/spec.md>

TypeScript : Ejercicio

1. Instalar de forma global typescript en nuestra máquina

```
npm install -g typescript
```

2. Crear un fichero .ts que contenga una clase persona con dos atributos:

1. Un atributo nombre de tipo string

2. Un atributo edad de tipo numérico

3. Un constructor que inicialice ambos atributos

4. Un método que imprima por pantalla (console.log(...)) el nombre y la edad de la persona

3. Utilizar el compilador de TypeScript para convertir nuestro fichero a código javascript

```
tsc myfile.ts
```

4. Observar el código javascript creado

Pista: Clase de ejemplo

TypeScript: Decoradores

- ▶ Soporte nativo de typescript (desde versión 2.x) para el uso de anotaciones
 - ▶ Permite añadir meta-information a nuestras clases
- ▶ Angular2 utiliza decoradores para la definición de todos sus elementos: módulos, componentes, servicios, etc.

```
@NgModule({  
    imports: [RouterModule.forRoot(routes)],  
    exports: [RouterModule],  
    providers: []  
})  
export class AppRoutingModule { }
```

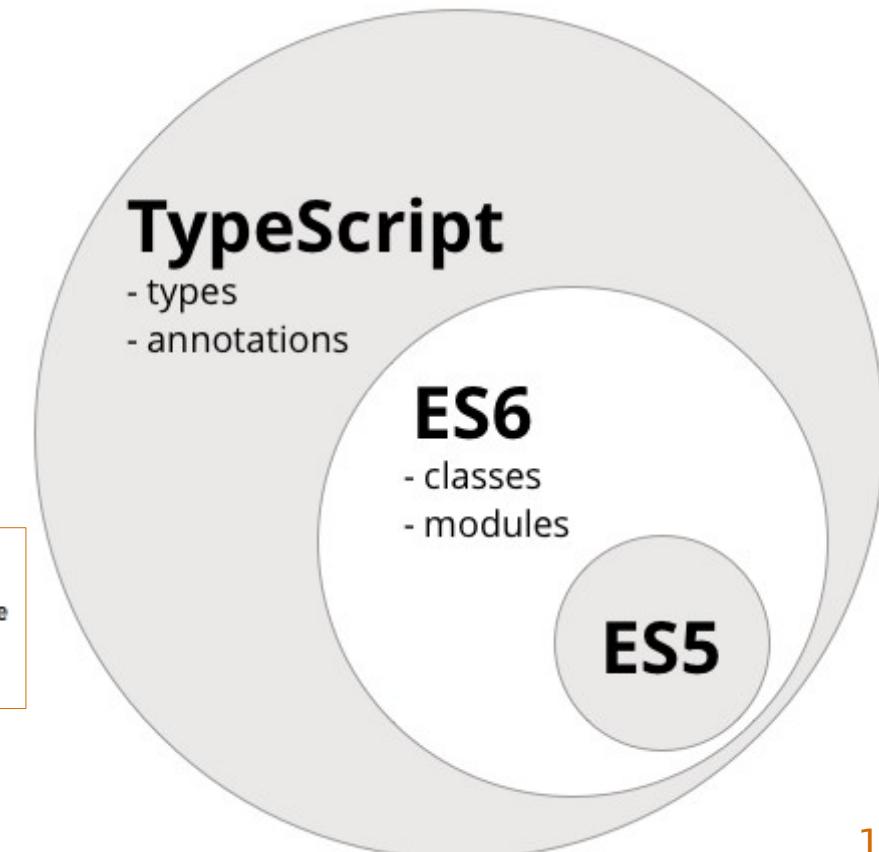
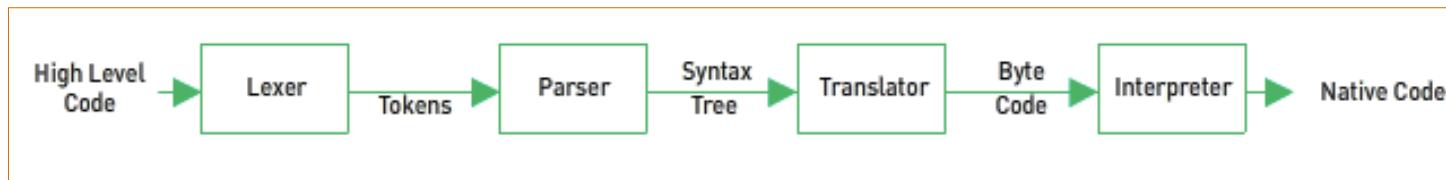
```
@Component({  
    selector: 'app-root',  
    templateUrl: './app.component.html',  
    styleUrls: ['./app.component.css']  
})  
export class AppComponent {  
    title = 'app works!';  
}
```

```
@Injectable()  
export class UserService {  
    private url: string = 'https://randomuser.me/api/?results=5';  
  
    constructor(private http: Http) {  
  
    }  
}
```

TypeScript, ES6 y ES5

- ▶ **ES5:** Javascript “tradicional”
- ▶ **ES6:** Javascript con clases, módulos, funciones flecha, etc ¹.
- ▶ **TypeScript:** *superset* de javascript. Soporta tipado y anotaciones

- ▶ ¿Cómo pasar de uno a otro? **Transpilers**
 - ▶ Angular2 utiliza el transpiler **tsc**
 - ▶ **Babel, traceur, etc.**



¹ <http://es6-features.org/>



Repaso de conceptos



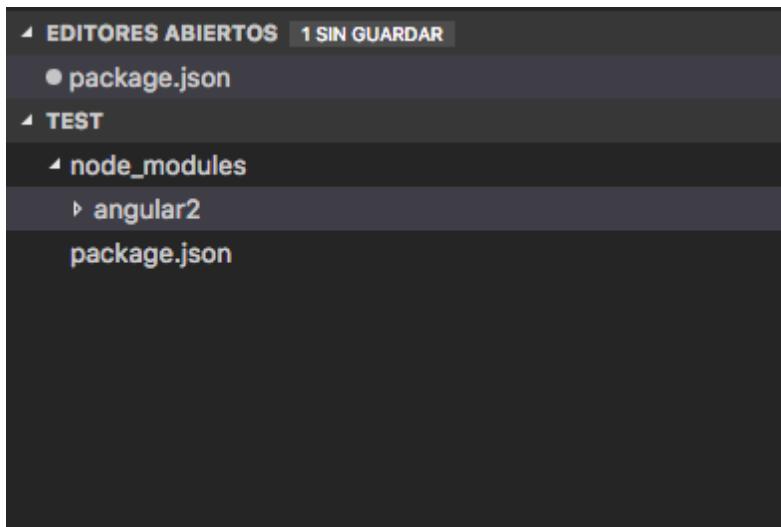
Anatomía de un proyecto Javascript : npm

- ▶ Gestor de dependencias para aplicaciones / librerías Javascript
- ▶ Un proyecto **npm** contiene un fichero `package.json` en el directorio raíz del proyecto
 - ▶ Ejemplo de package.json
- ▶ npm gestiona las dependencias de nuestro proyecto, tareas de desarrollo, lista de contribuyentes, etc.
- ▶ Sin **npm**, los proyectos 100% Javascript eran gestionados de forma manual: las dependencias eran descargas y guardadas manualmente.
- ▶ Con **npm**, podemos especificar la lista de dependencias en un fichero de texto y hacer un `npm install` para descargar todas ellas
- ▶ Es utilizado en proyectos front-end y back-end (NodeJS)
- ▶ Algunos de los proyectos que utilizan NPM... (el 99,9% de proyectos JS):
 - ▶ Angular2 : <https://github.com/angular/angular/blob/master/package.json>
 - ▶ D3 : <https://github.com/d3/d3/blob/master/package.json>
 - ▶ React : <https://github.com/facebook/react/blob/master/package.json>



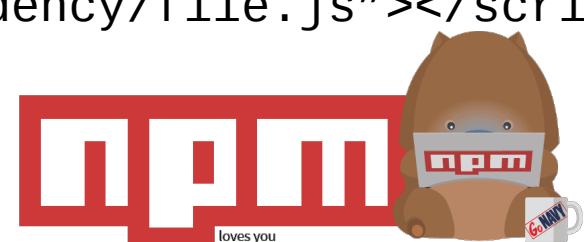
Anatomía de un proyecto Javascript : npm

- A continuación se relatan los pasos necesarios para crear un proyecto npm:
 1. npm init # (Completar información)
 2. npm install angular2 -save



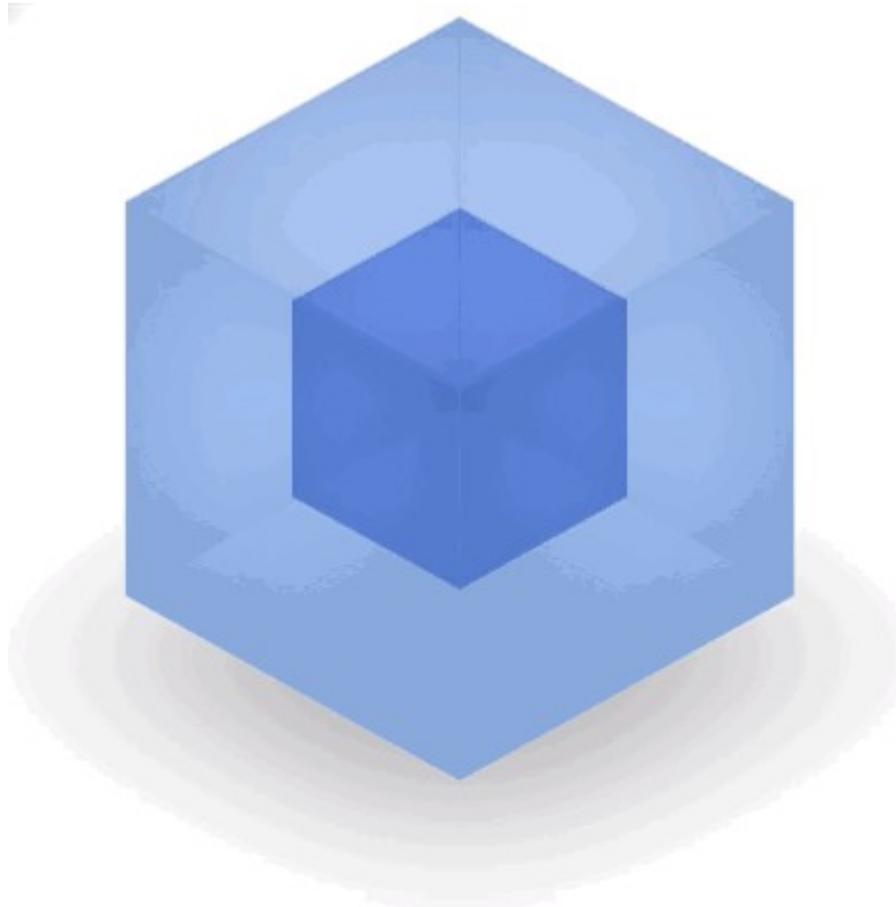
```
1  {
2    "name": "test",
3    "version": "1.0.0",
4    "description": "",
5    "main": "index.js",
6    "scripts": {
7      "test": "echo \\\"Error: no test specified\\\" && exit 1"
8    },
9    "author": "",
10   "license": "ISC",
11   "dependencies": {
12     "angular2": "^2.0.0-beta.21"
13   }
14 }
15 }
```

```
<script type="text/javascript" src=".node_modules/myDependency/file.js"></script>
```



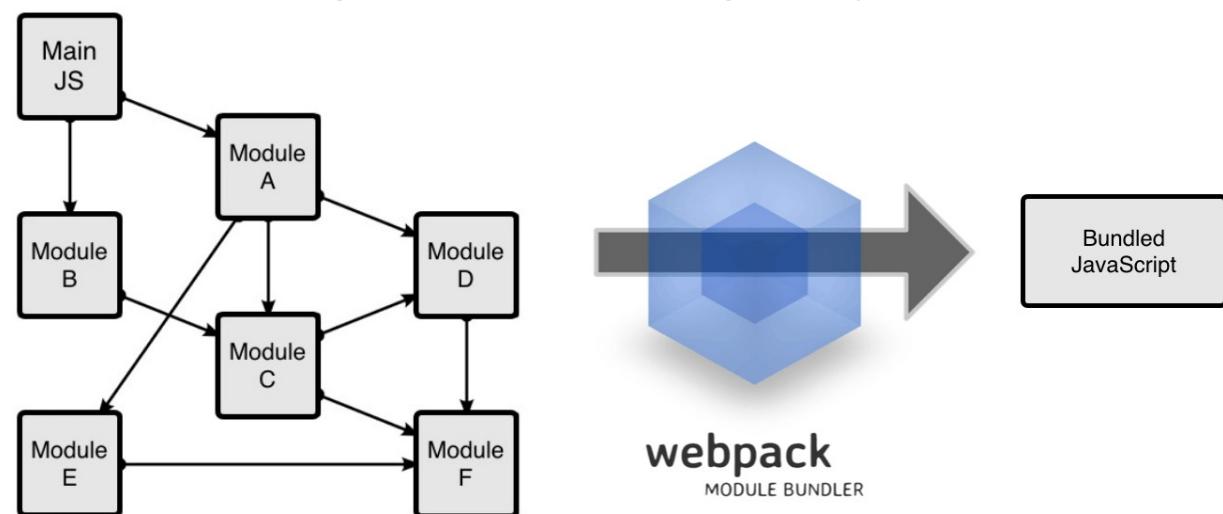


Repaso de conceptos

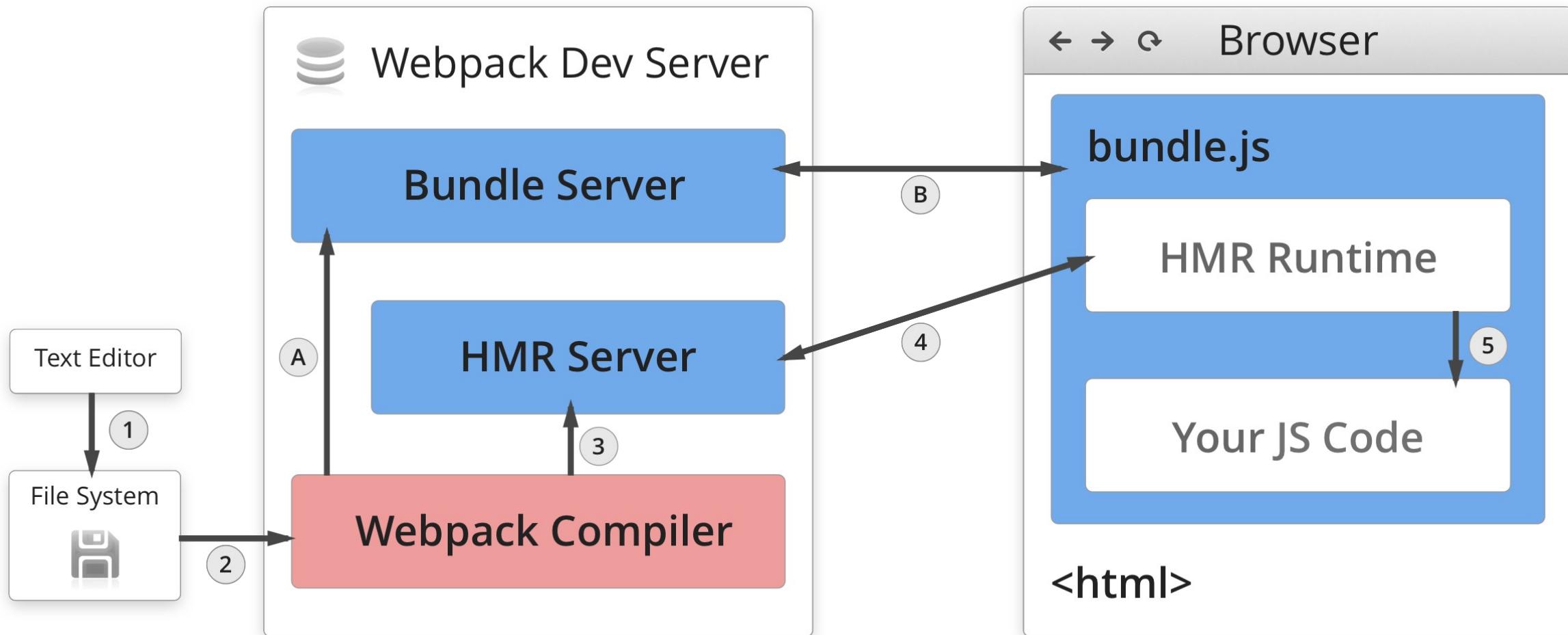


Webpack

- ▶ Open-source Javascript **module bundler**
- ▶ Crea **assets estáticos** a partir de un proyecto modularizado con dependencias
 - ▶ Genera un grafo de dependencias permitiéndonos usar un enfoque modularizado en nuestras aplicaciones web
- ▶ Funciona con node.js
- ▶ Es **altamente extensible**: podemos incorporar *loaders* que ejecuten tareas personalizadas



Webpack



Webpack: Ejemplo

example.js

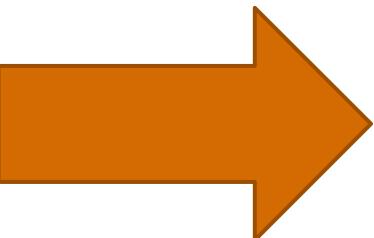
```
var inc = require('./increment').increment;  
var a = 1;  
inc(a); // 2
```

math.js

```
exports.add = function() {
    var sum = 0, i = 0, args = arguments, l = args.length;
    while (i < l) {
        sum += args[i++];
    }
    return sum;
};
```

increment.js

```
var add = require('./math').add;
exports.increment = function(val) {
    return add(val, 1);
};
```



```
***** / ([  
/* 0 */  
/* unknown exports provided */  
/* all exports used */  
/*!*****!*\
  !*** ./increment.js ***!
  \*****/  
/**/ function(module, exports, __webpack_require__) {  
  
var add = __webpack_require__(/*! ./math */ 1).add;  
exports.increment = function(val) {  
    return add(val, 1);  
};  
  
/**/ },  
/* 1 */  
/* unknown exports provided */  
/* all exports used */  
/*!*****!*\
  !*** ./math.js ***!
  \*****/  
/**/ function(module, exports) {  
  
exports.add = function() {  
    var sum = 0, i = 0, args = arguments, l = args.length;  
    while (i < l) {  
        sum += args[i++];  
    }  
    return sum;  
};  
  
/**/ },  
/* 2 */  
/* unknown exports provided */  
/* all exports used */  
/*!*****!*\
  !*** ./example.js ***!
  \*****/  

```

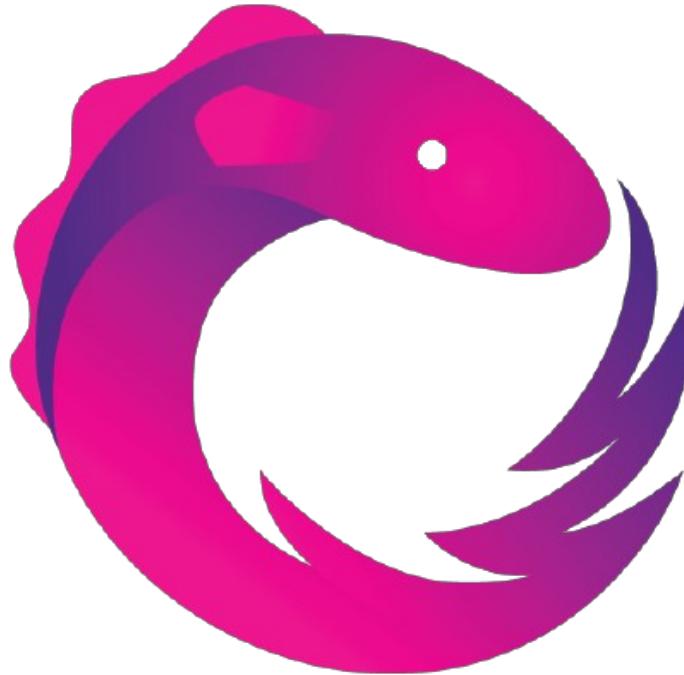
TypeScript & Webpack: Proyecto de ejemplo

<https://github.com/proteus-h2020/proteic>





Repaso de conceptos

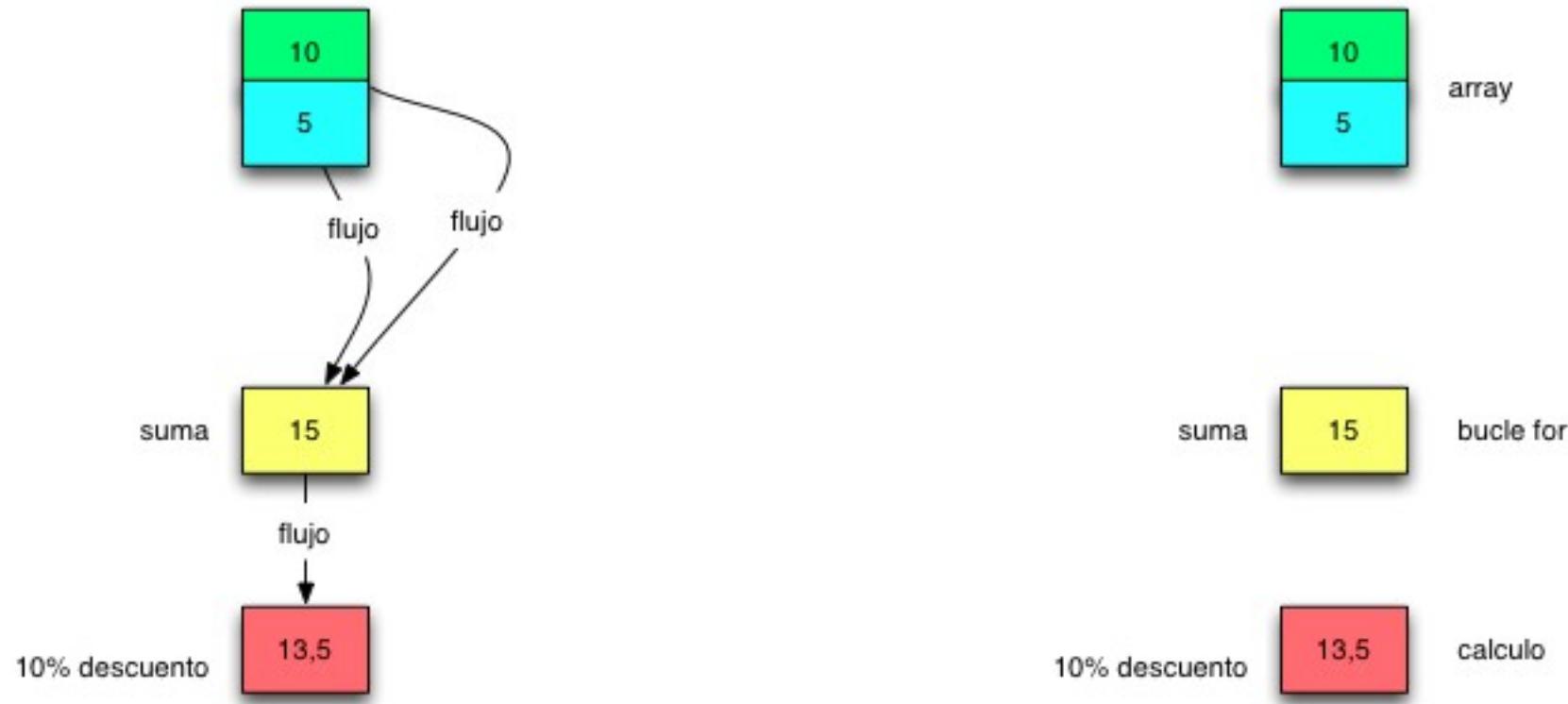


Programación reactiva funcional

- ▶ Paradigma de programación orientada a los ***data flows*** (flujos de datos) y la **propagación de sus cambios**
 - ▶ Todos los datos se consideran *streams* o *flujos*
- ▶ Genera un código altamente reutilizable
- ▶ Características:
 - ▶ **Responsividad:** útil, usable. Simplifica el número y manejo de errores en nuestra aplicación.
 - ▶ **Resiliencia:** aplicación consciente y “viva” en caso de fallos. Aislamiento de errores.
 - ▶ **Elasticidad:** Diseño sin cuellos de botella.
 - ▶ **Orientado a mensajes de datos:** Todo son flujos de datos asíncronos. No existe nada bloqueante.



Reactivo VS No reactivo



Programación NO reactiva (Listener)

```
<!DOCTYPE>
<html>

<head>
    <title></title>
    <script src="rx.all.js" type="text/javascript">
        </script>

</head>

<body>
    <input type="button" id="boton" value="pulsar" />
</body>
<script type="text/javascript">
    var total = 0;

    var boton = document.getElementById("boton");
    boton.addEventListener("click", function () {

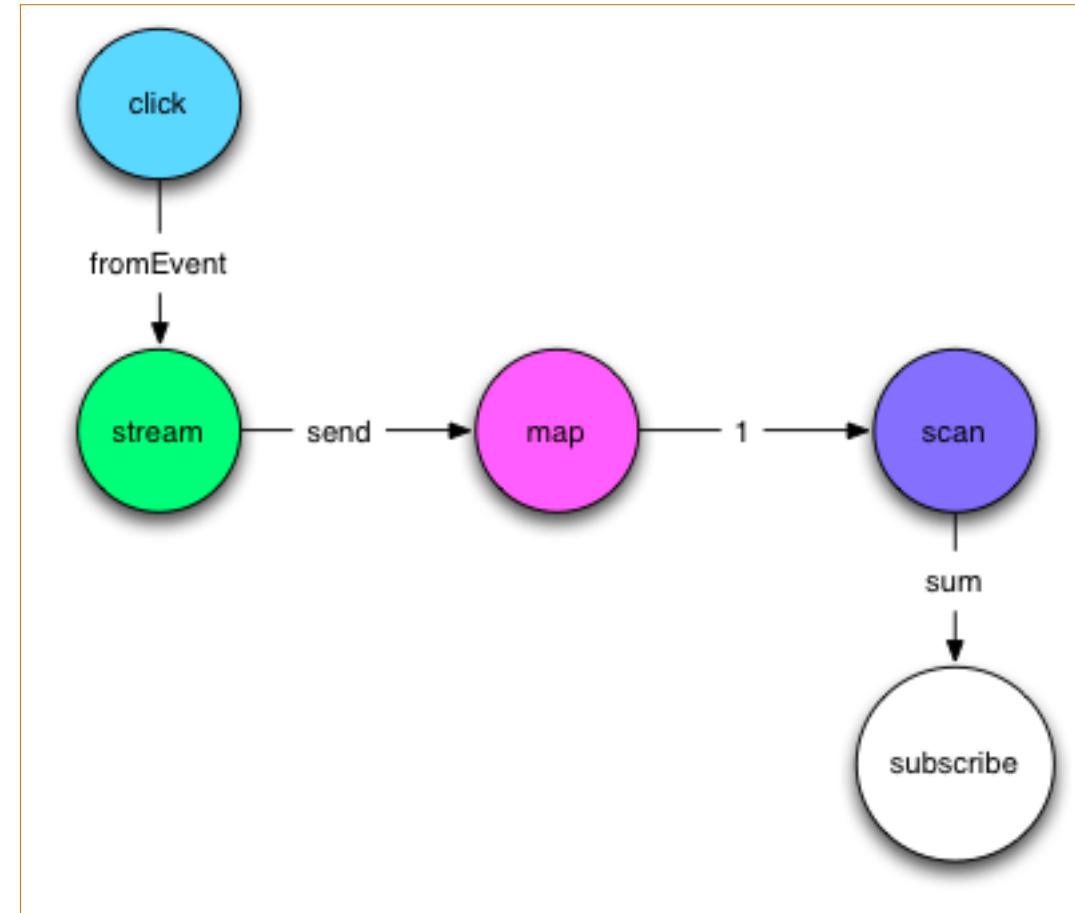
        console.log(++total);
    });
</script>

</html>
```

Programación reactiva con RxJS

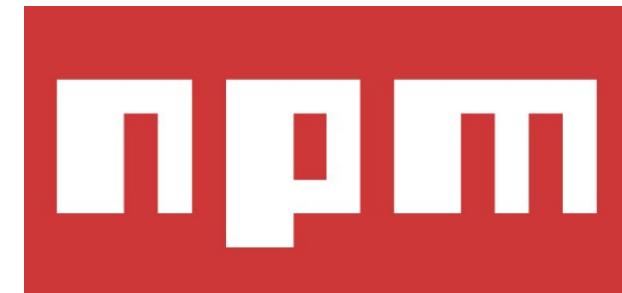
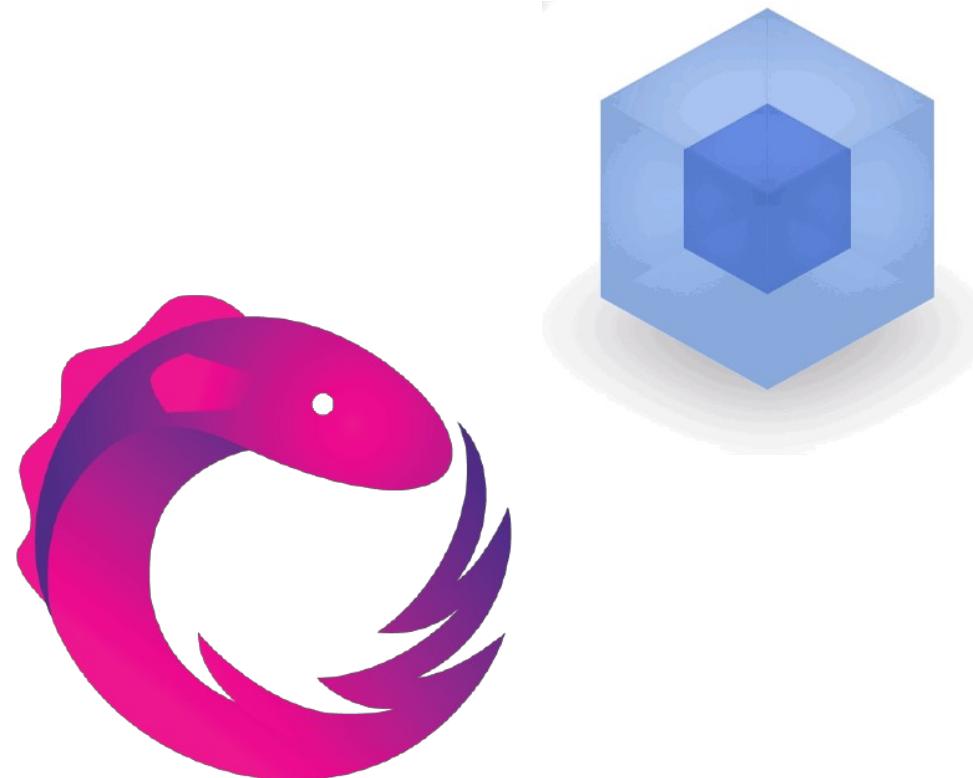
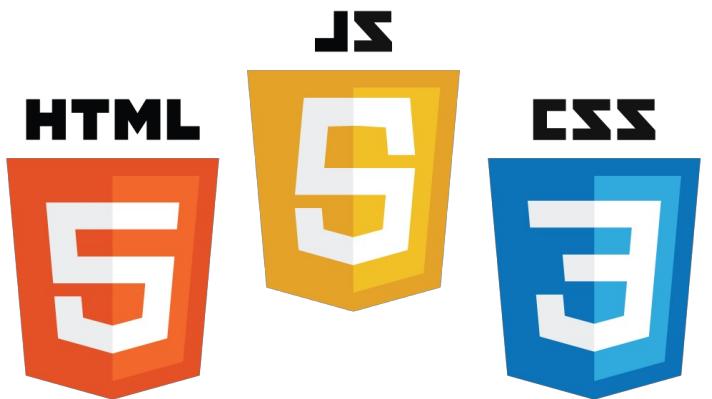
```
<html>
<head>
<title></title>
<script src="rx.all.js" type="text/javascript">
</script>
</head>
<body>
<input type="button" id="boton" value="pulsar" />
</body>
<script type="text/javascript">
var boton = document.getElementById("boton");
var botonStream = Rx.Observable.fromEvent(boton, 'click');
var suma=botonStream
.map(function(x) {
  return 1;
}).scan(function(x,y) {
  return x+y;
}, 0).subscribe(function(resultado) {
  console.log(resultado);
});
</script>

</html>
```





Repaso de conceptos





Introducción a Angular



1957



2017



2010



2017



¿Quieres una introducción a AngularJS 1?

<http://www.slideshare.net/0xnacho/angularjs-1-spanish>



¿Por qué Angular2?

Multiplataforma



Rápido



Escalable



AngularJS / Angular

- AngularJS es un framework de desarrollo MVC destinado al desarrollo de aplicaciones front-end SPA o *Single Page Application*.

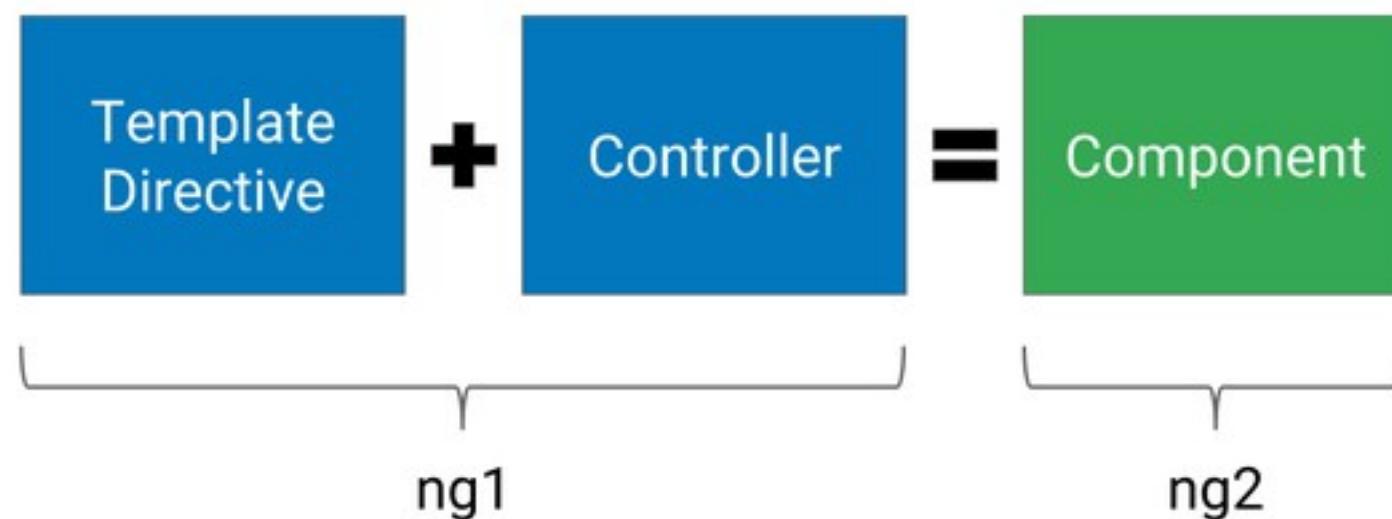


- Angular es una reimplementación de AngularJS destinada a mejorar todas las carencias o puntos débiles de AngularJS:
 - Aplicaciones complejas
 - Tiempo en renderizado de plantillas
 - Gestión de datos y servicios
 - Inyección de dependencias

Diferencias: Angular1 vs Angular2

	AngularJS	Angular2
Rendimiento		5 veces más rápido
Lenguajes (APIs)	Javascript ES5	Javascript ES5, ES6, Typescript
Arquitectura	Modular programming	Component-oriented
Directivas	Incontables	Component, Decorator, Template
Servicios	Factorías, Servicios, Proveedores, Variables, Constantes	Injectable
Soporte móvil en mente	No	Sí
Sintaxis	Compleja	Sencilla
Detección de cambios	Cycle digest	Zone.js

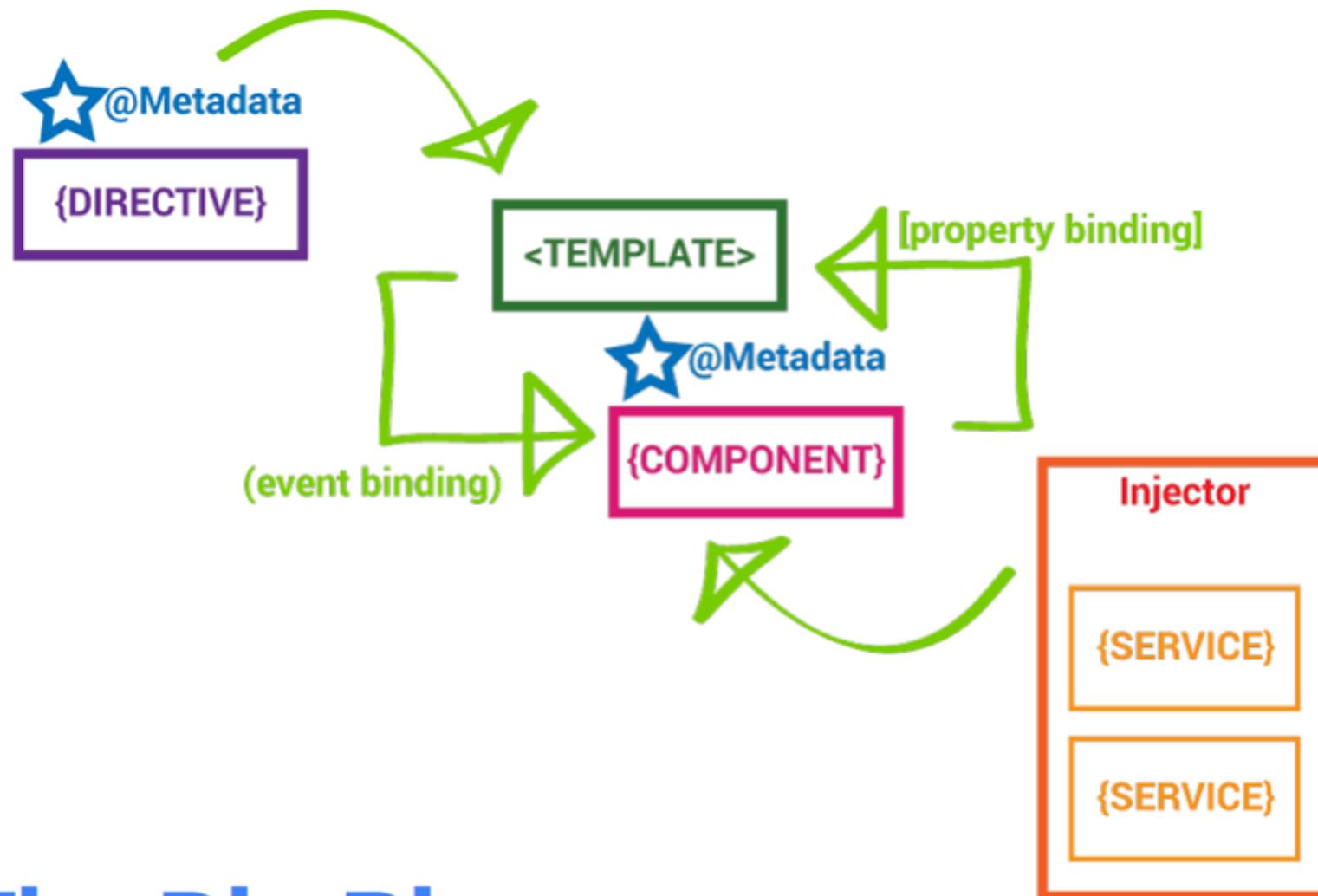
Diferencias: Angular1 vs Angular2



Diferencias: Angular1 vs Angular2



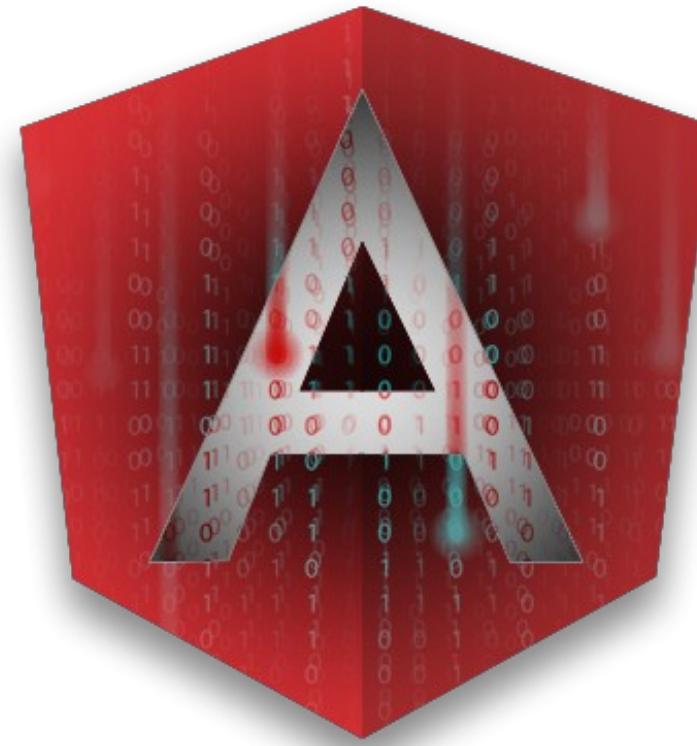
Angular2 : Arquitectura



The Big Picture

Angular2 : Conceptos fundamentales

- ▶ Módulos
- ▶ Componentes
- ▶ Metadatos
- ▶ Templates
- ▶ Data Binding
- ▶ Servicio
- ▶ Directiva
- ▶ Inyección de dependencias

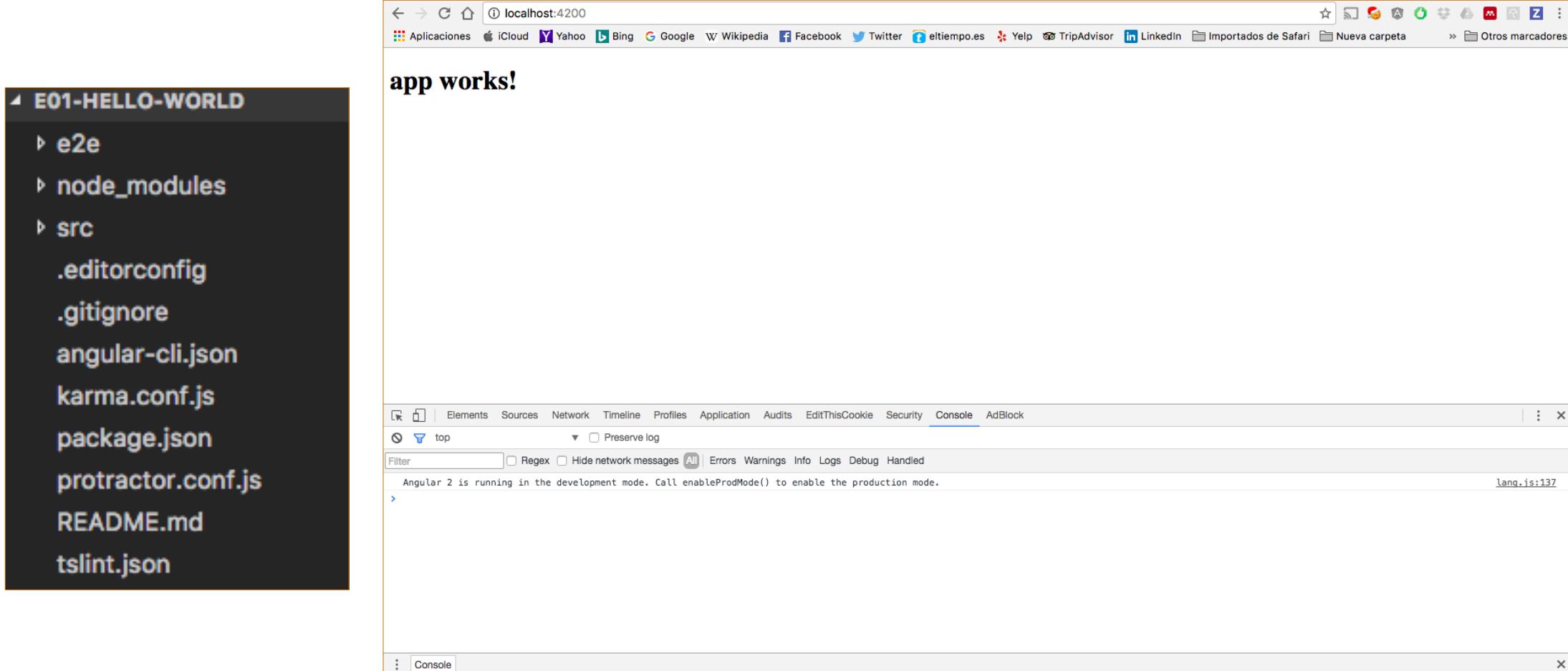


Angular2 CLI

- ▶ Herramienta de comandos proporcionada por Angular2
 - ▶ Nos facilita la tareas de configuración inicial de nuestro proyecto (evitamos la configuración de webpack, typescript, etc).
- ▶ Permite generar proyectos Angular2 y añadir funcionalidades a nuestro proyecto desde la terminal
- ▶ Para instalar la herramienta de forma global en nuestro sistema, introduciremos el siguiente comando:
 - ▶ `npm install -g angular-cli # -g instala de forma global`
- ▶ Posteriormente, crearemos un proyecto angular2:
 - ▶ `ng new E01-hello-world`
- ▶ Ahora probaremos que la app funciona correctamente:
 - ▶ `cd E01-hello-world`
 - ▶ `ng serve`



Angular2 CLI

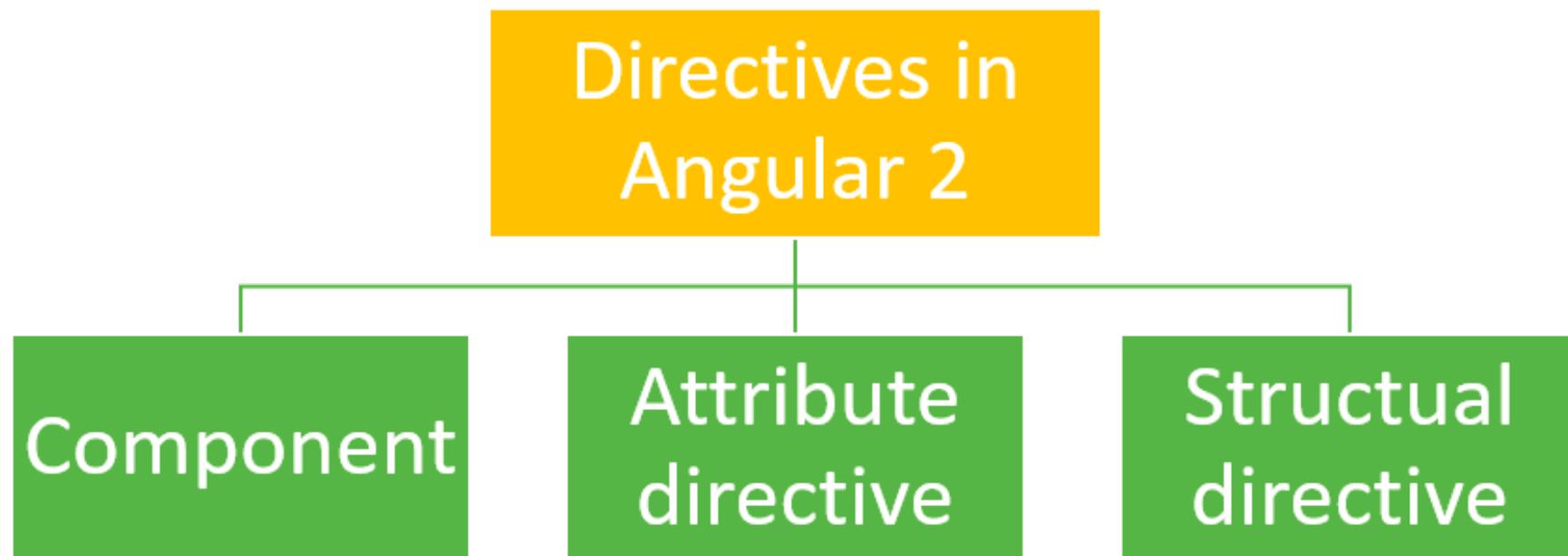


Angular2 CLI : Opciones adicionales

- ▶ **ng serve:** sirve nuestra app en un servidor web ligero:
 - ▶ --host: interfaz de red desde la que es accesible la aplicación
 - ▶ --port: puerto en el que trabajará el servidor web (por defecto 4200)
 - ▶ --live-reload-port: puerto en el que trabaja el servidor *live-reload*
 - ▶ --proxy-config: fichero de configuración proxy
- ▶ **ng test:** ejecuta los tests unitarios del proyecto
- ▶ **ng e2e:** ejecuta los tests de extremo a extremo
- ▶ **ng build:** construye el proyecto en dist/:
 - ▶ --target: production o development

Scaffold	Usage
Component	<code>ng g component my-new-component</code>
Directive	<code>ng g directive my-new-directive</code>
Pipe	<code>ng g pipe my-new-pipe</code>
Service	<code>ng g service my-new-service</code>
Class	<code>ng g class my-new-class</code>
Interface	<code>ng g interface my-new-interface</code>
Enum	<code>ng g enum my-new-enum</code>
Module	<code>ng g module my-module</code>

Angular2 : Directivas





Módulos



Módulos

- ▶ Se define módulo como una agrupación **conjunto de código fuente relacionado entre sí**
 - ▶ Contiene una serie de componentes y servicios
- ▶ El concepto de módulo existe desde la primera versión de AngularJS

```
var module = angular.module('myApp.services', []);  
  
module.service('version', function(){  
    // Public API  
    return {  
        getVersion: function(){  
            return '0.1';  
        }  
    };  
});  
  
module.constant('version', '0.1');  
  
// other possible service declarations:  
module.factory(...);  
module.provider(...);  
module.value(...);
```

Declare a Service API

Declare a Constant

Different Declaration Styles
for different use cases

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

AngularJS

Angular

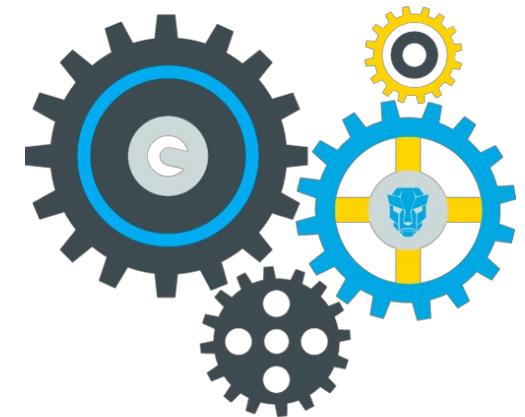
Módulos

- ▶ El decorador `@NgModule` utiliza una serie de parámetros:
 - ▶ **declarations**: Componentes que pertenecen al módulo actual
 - ▶ **imports**: Módulos utilizados por nuestro módulo (e.g. puede ser el caso de que necesitemos utilizar componentes o servicios pertenecientes a otros módulos. En tal caso, deberemos importar el módulo entero)
 - ▶ **providers**: Servicios utilizados por los componentes de este módulo
 - ▶ **exports**: Define que componentes exportar para ser importados por otros módulos
 - ▶ **bootstrap**: Componente que inicia la aplicación. Idealmente deberíamos tener un único componente root

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



Componentes



Componentes

- ▶ Una aplicación en Angular2 está **compuesta** principalmente por **componentes**
- ▶ Un componente es la **combinación de una plantilla HTML** y **una clase** en Typescript que controla su funcionamiento.
 - ▶ Un componente en Angular2 = Controlador + Directiva en Angular1
- ▶ Los componentes son declarados con **Decorators**¹
 - ▶ Añaden meta información a una clase de Typescript
- ▶ Los componentes deberán seguir la siguiente nomenclatura:
`<nombre>.component.ts`

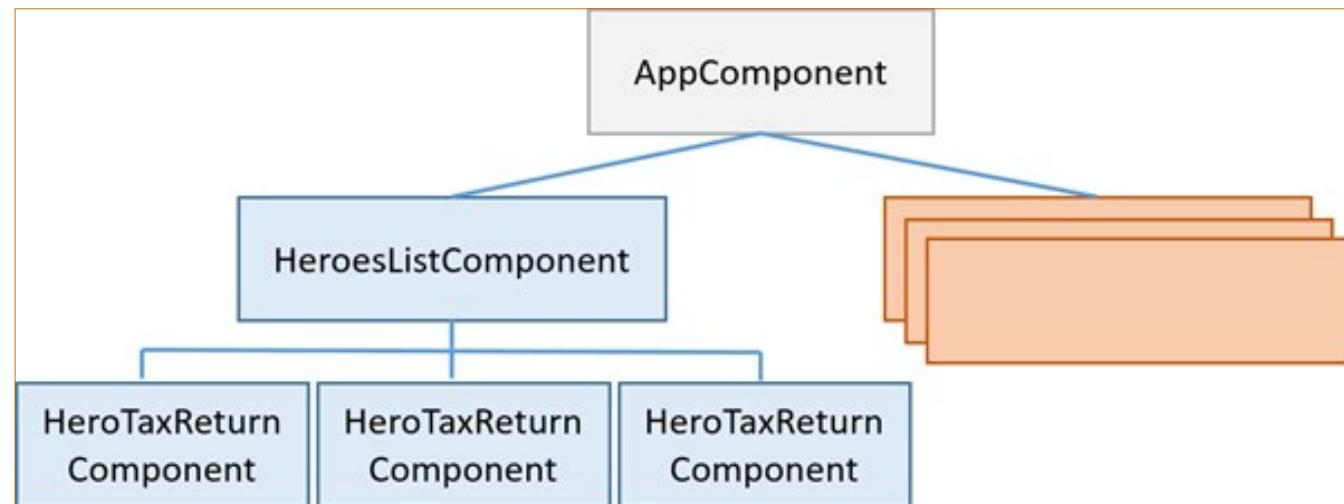
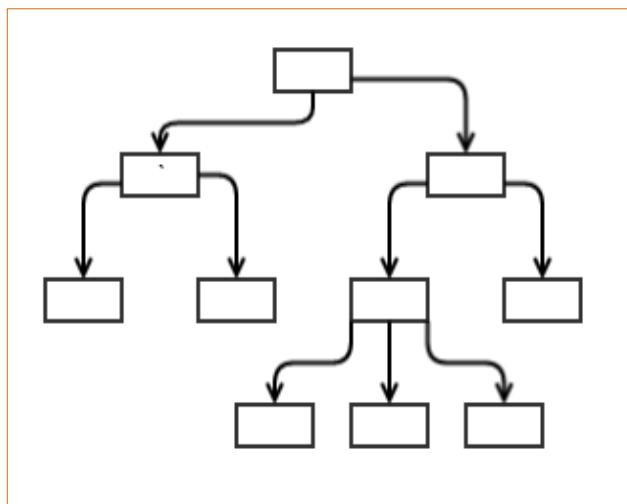
```
import { Component } from '@angular/core';

@Component({
  selector: 'my-app',
  template: `<h1>Hello {{name}}</h1>`
})
export class AppComponent { name = 'Angular'; }
```

¹ <https://angular.io/docs/ts/latest/glossary.html#!#decorator>

Componentes: Jerarquía

- ▶ Existirá **un componente global**, padre del árbol de componentes de nuestro módulo
 - ▶ Este componente será el indicado en la opción **bootstrap** de nuestro módulo
- ▶ A partir del componente inicial generaremos un **árbol con el resto de componentes**
 - ▶ Cada uno de los hijos puede tener más hijos, y así sucesivamente



Componentes: Parámetros del decorador

- ▶ **selector**: Nombre del tag HTML con el que se invocará al componente
- ▶ **template**: Código HTML de la plantilla
- ▶ **templateUrl**: Path al documento HTML de este componente
- ▶ **styles**: Estilos CSS de este componente
- ▶ **styleUrl**: Path al documento .css de este componente

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

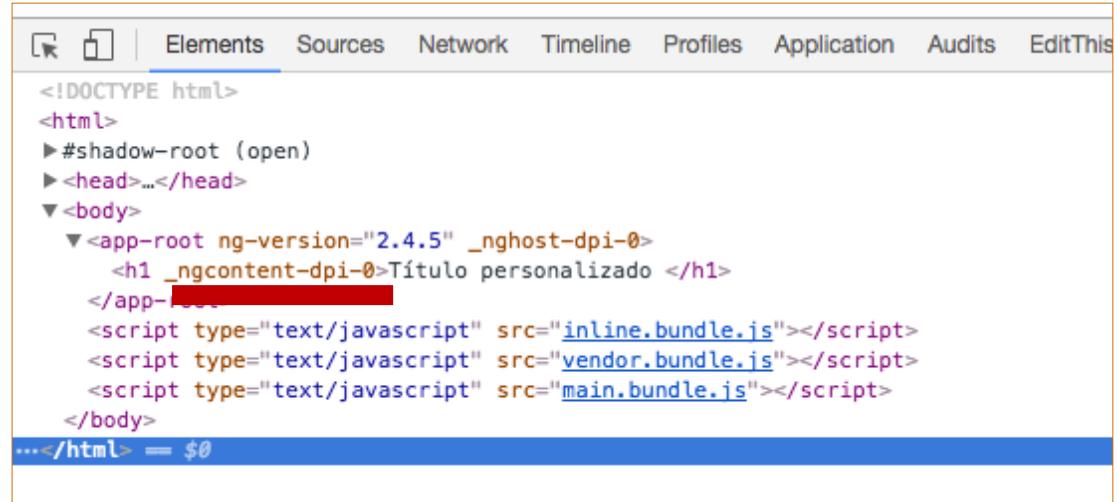
```
@Component({  
  selector: 'app-root',  
  template: '<h1>Hey </h1>',  
  styles: ['table { background: green; }'],  
})  
export class AppComponent {  
  title = 'app works!';  
}
```

Componentes: Estilos

- ▶ ¡Importante!: Los **estilos** que damos a nuestros elementos en un **componente** solo **afectan** a los elementos de ese **componente**
- ▶ Si queremos declarar un estilo para un elemento de forma global, utilizaremos el fichero src/styles.css
- ▶ Si hay conflicto, podemos utilizar la keyword !important en nuestro CSS

```
@Component({  
  selector: 'app-root',  
  template: '<h1>Título personalizado </h1>',  
  styles: ['h1 { color: orange; }'],  
})  
export class AppComponent {  
  title = 'app works!';  
}
```

Título personalizado



Componentes: Invocación

- ▶ Para invocar al componente, bastará con introducir <my-app></my-app> en un fichero HTML.
 - ▶ El contenido de este tag es lo que se mostrará hasta la carga completa del componente (0s-1s):
 - ▶ <my-app> Cargando contenido.... </my-app>
 - ▶ Una vez cargado el contenido, el tag <my-app> es sustituido por el contenido real de la aplicación (el definido en la plantilla)

```
<!doctype html>
<html>
<head>
  <meta charset="utf-8">
  <title>E01HelloWorld</title>
  <base href="/">

  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="icon" type="image/x-icon" href="favicon.ico">
</head>
<body>
  <app-root>Loading...</app-root>

</body>
</html>
```

Componentes: Creación

- ▶ **Sencillo:**
 - ▶ `ng g component <NOMBRE_COMPONENTE>`
- ▶ **Elaborado:**
 - ▶ Crear una carpeta con el nombre del componente dentro de `src/app`
 - ▶ Crear los ficheros del componente:
 - ▶ `<NOMBRE_COMPONENTE>.component.html`
 - ▶ `<NOMBRE_COMPONENTE>.component.css`
 - ▶ `<NOMBRE_COMPONENTE>.component.ts`
 - ▶ `<NOMBRE_COMPONENTE>.component.spec.ts`
 - ▶ Editar el fichero de configuración de nuestro módulo para incluir nuestro componente:
 - ▶ Añadir nuestro componente en el array de declarations dentro de `app.module.ts` (o cualquier otro módulo de nuestra aplicación)

Componentes: Creación

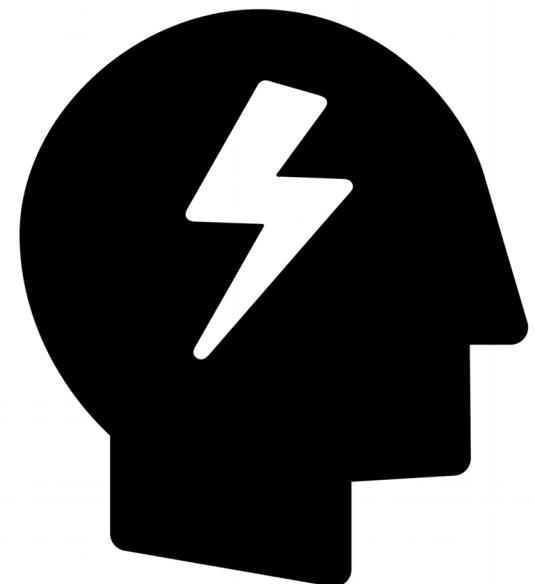
- ▶ Crearemos el link entre los diferentes archivos, añadiendo el decorador a nuestro <NOMBRE_COMPONENTE>.component.ts
- ▶ Enlazaremos nuestra hoja de estilos mediante el campo styleUrls[]
- ▶ Enlazaremos nuestra vista HTML mediante el campo template
- ▶ Indicaremos el selector que utilizaremos para invocar el componente mediante el campo selector

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

```
@Component({  
  selector: 'app-mytest',  
  templateUrl: './mytest.component.html',  
  styleUrls: ['./mytest.component.css']  
})  
export class MytestComponent implements OnInit {  
  
  constructor() {}  
  
  ngOnInit() {}  
}
```

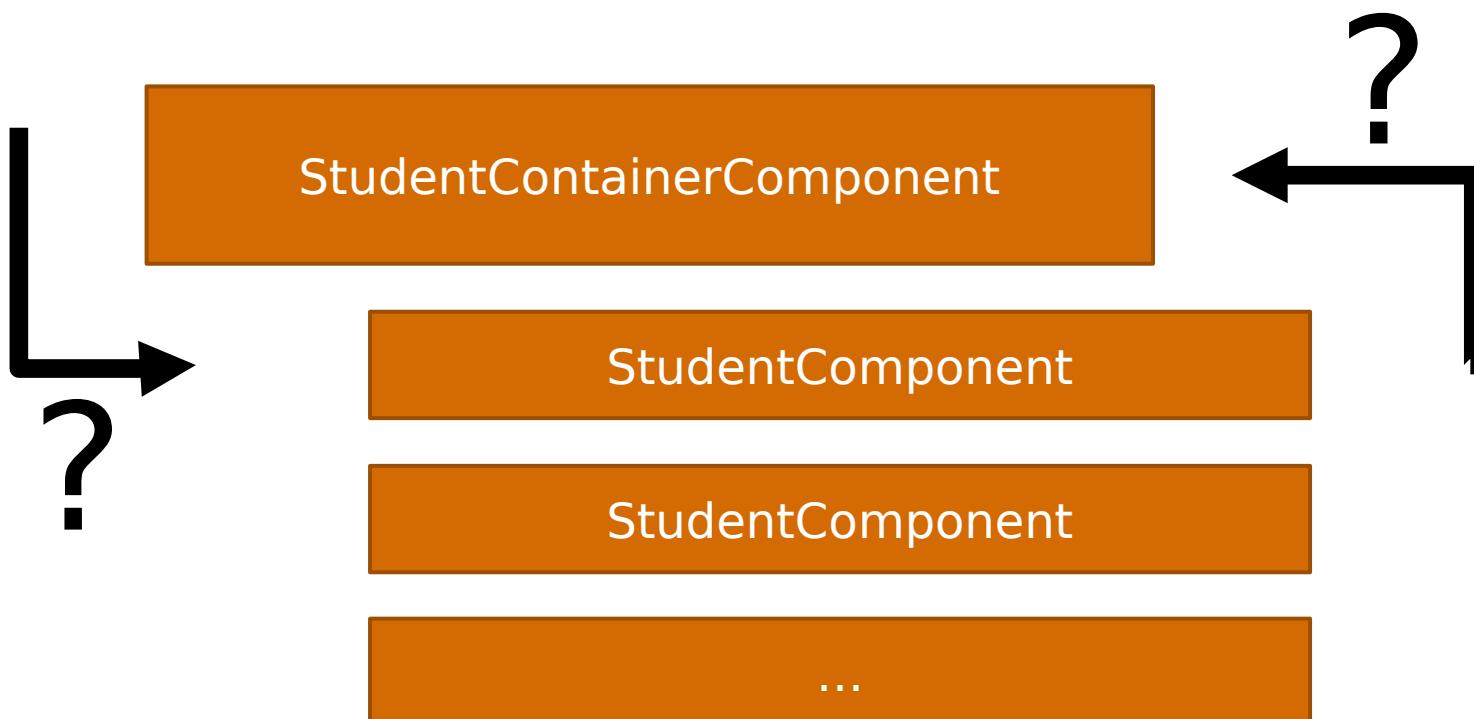
Componentes: Ejercicio

- ▶ Examinar paso a paso el funcionamiento y lógica del proyecto creado previamente (**E01-hello-world**)
- ▶ Añadir un nuevo componente al proyecto (PersonComponent) que muestre:
 - ▶ Tu nombre
 - ▶ Tus apellidos



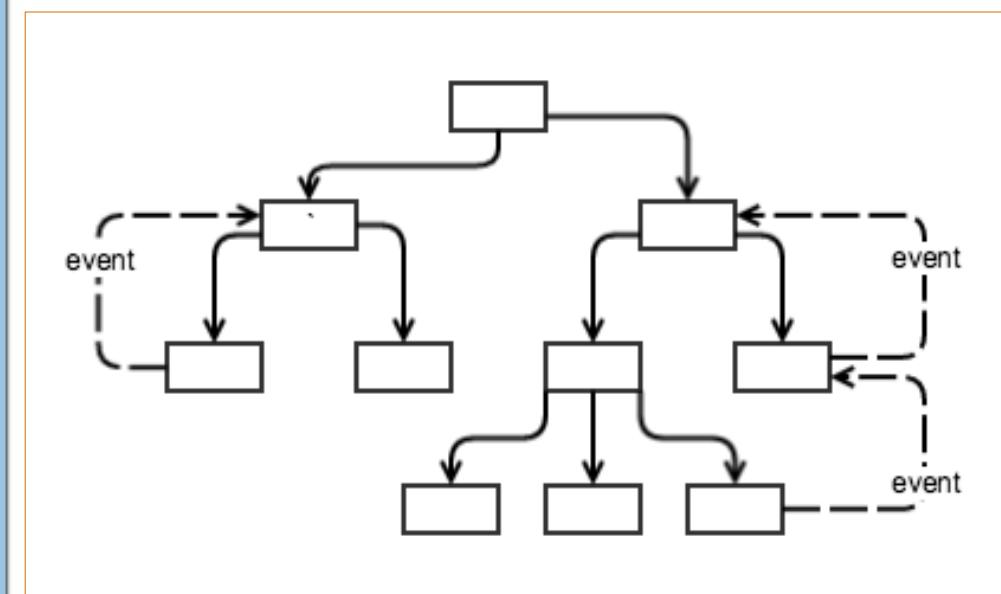
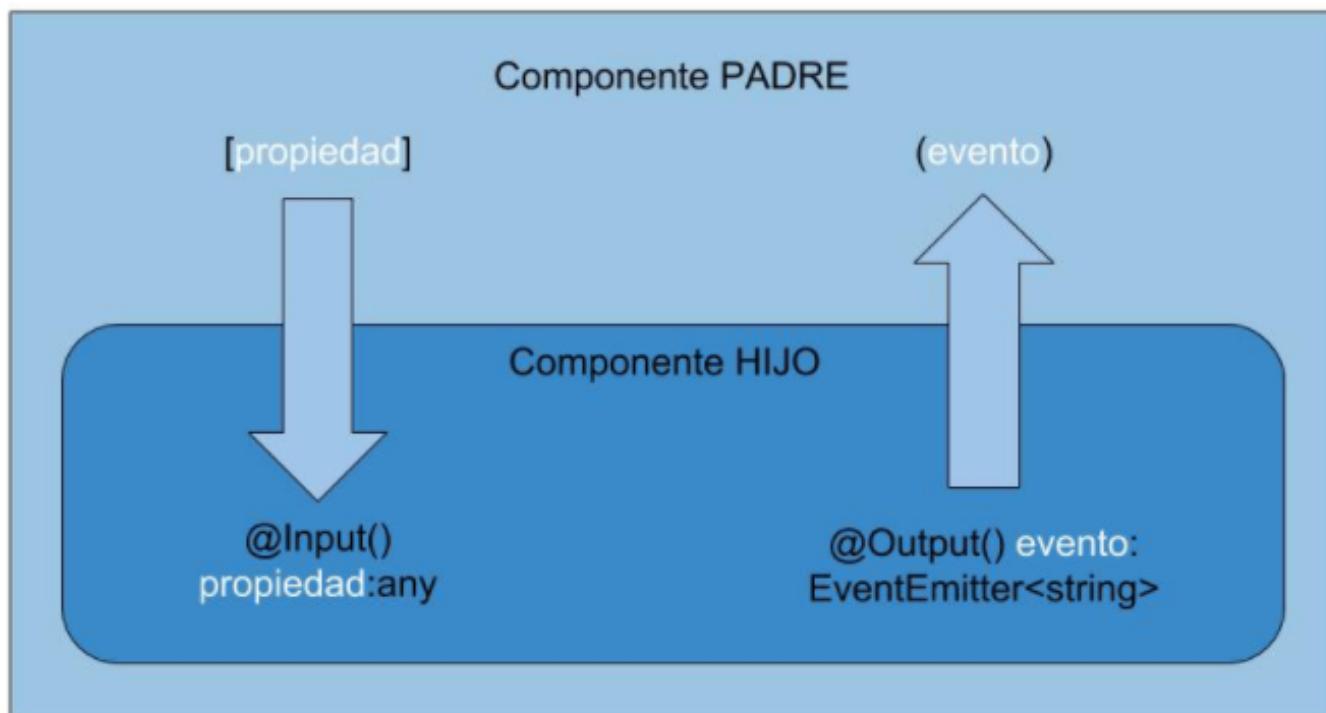
Componentes: Comunicación

- ▶ Cada componente tiene su funcionalidad encapsulada en su fichero <nombre>.component.ts
- ▶ ¿Cómo podemos **intercambiar datos** entre un componente **padre** y su **hijo**?



Componentes: Comunicación

- Angular nos proporciona los decoradores `@Input` y `@Output`. Los `@Input` envían datos hacia componentes hijos y los `@Output` envían eventos al padre.



Componentes: @Input

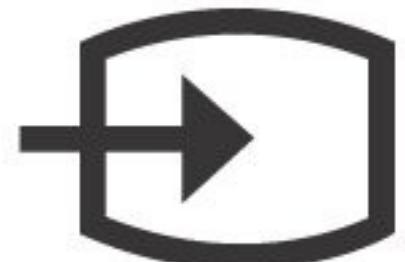
- ▶ Mecanismo para enviar datos de un componente padre a un hijo
- ▶ El envío de los datos se realiza **a través de la vista HTML del componente padre:**
 - ▶ Cuando invoquemos el componente hijo desde nuestro padre, indicaremos un **atributo con un nombre y un valor**
 - ▶ Este valor puede ser interpretado ([]) o directo
 - ▶ Dentro del componente hijo, creamos un atributo con el decorador @Input('nombre_variable') el cual recoja el valor del padre

```
<counter [counterValue]=""myValue"></counter>  
  
<counter counterValue=""myValue"></counter>
```

parent.component.html

```
@Component({})
export class CounterComponent {
  @Input() counterValue = 0;
  increment() {
    this.counterValue++;
  }
  decrement() {
    this.counterValue--;
  }
}
```

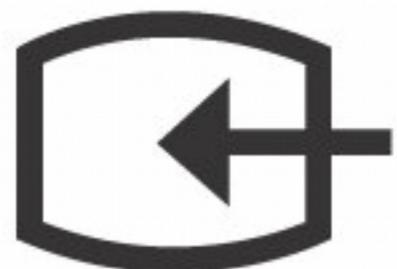
children.component.ts



Componentes: @Output

- ▶ Mecanismo para enviar eventos desde un hijo a su padre
- ▶ El **componente hijo** deberá tener **un atributo de tipo EventEmitter¹:**
 - ▶ Deberemos anotar este atributo con el decorador `@Output('name')`
 - ▶ Los datos serán enviados al componente padre cuando se llame al evento `this.emitter.emit(data)`
- ▶ Un output es expresado entre paréntesis
- ▶ Las acciones de los outputs pueden quedar internamente en el componente asociado, o comunicarse con otros (depende del fin de la acción).
- ▶ Algunos de los más típicos son: `(click)`, `(dblclick)`,

```
<counter [counterValue]="counterValue" (change)="countChange($event)"></counter>
```



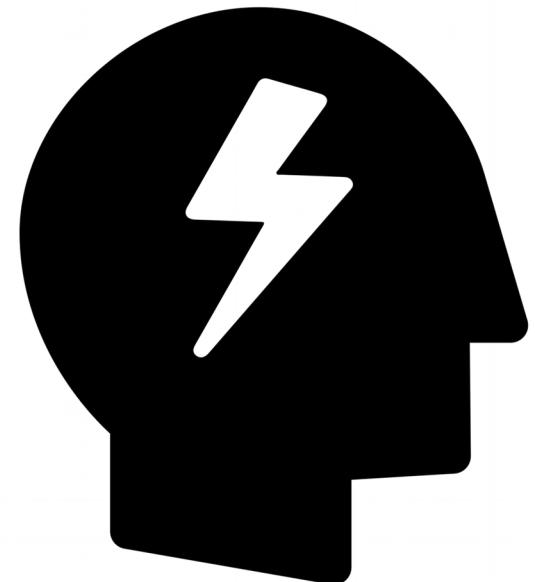
¹ <https://angular.io/docs/ts/latest/api/core/index/EventEmitter-class.html>

Componentes: Ejemplo @Input y @Output

<http://plnkr.co/edit/VVY9tywH3Zxm4TmWR86R?p=preview>

Componentes: Ejercicio

- ▶ Crear una copia del ejercicio anterior y renombrarlo a **E01-hello-world_input-output**
- ▶ La aplicación realizará la misma función (mostrar nombre y apellidos de la persona), pero en este caso los datos nombre y apellidos serán recibidos del componente padre (AppComponent), en lugar de instanciarse en la propia clase
- ▶ **Al pasar por encima del nombre** se enviará una notificación al componente padre, el cual imprimirá por consola (console.log(...)) un mensaje (un valor cualquiera enviado por el hijo).



Componentes: Ciclo de vida

- ▶ Las instancias de componentes y directivas tienen un ciclo de vida gestionado por Angular2:
 - ▶ Se crean, actualizan y destruyen
- ▶ Los desarrolladores tienen acceso a este ciclo de vida mediante la implementación de una serie de interfaces
 - ▶ Cada una de estas interfaces tiene un único método (i.e: OnInit -> ngOnInit())

constructor

ngOnChanges

ngOnInit

ngDoCheck

ngAfterContentInit

ngAfterContentChecked

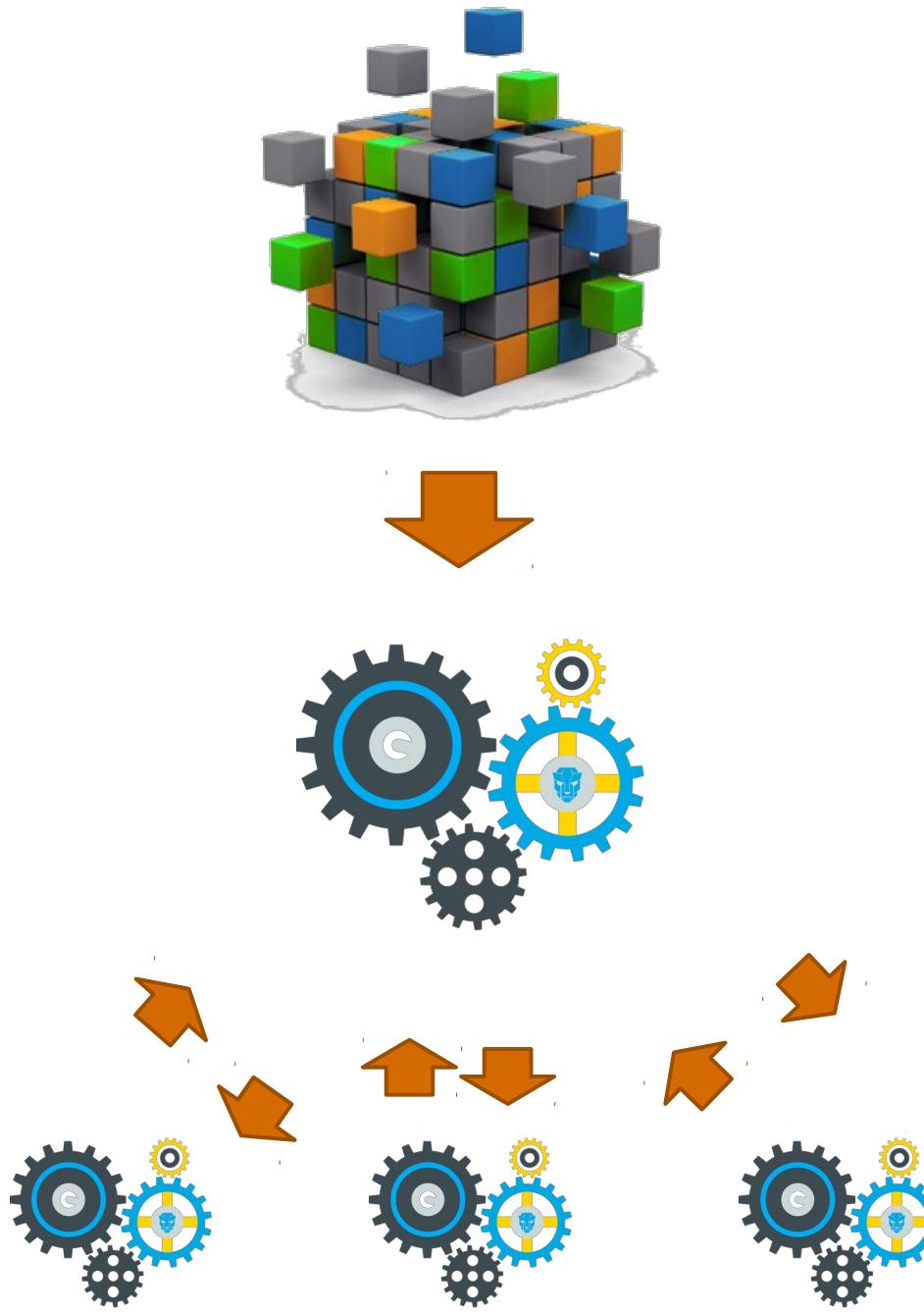
ngAfterViewInit

ngAfterViewChecked

ngOnDestroy

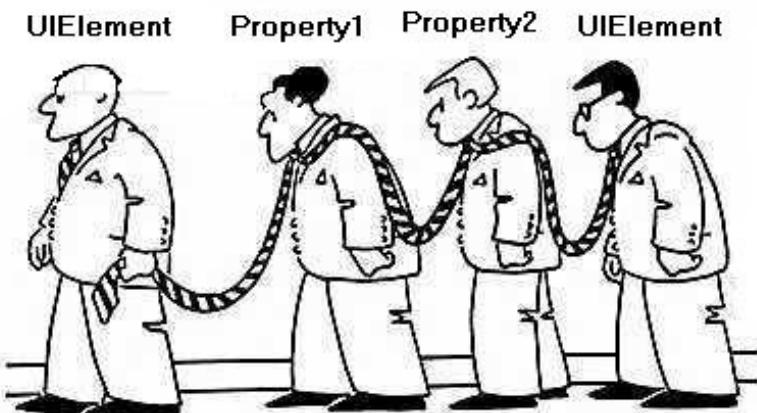
Componentes: Interfaces complementarias

- ▶ OnInit: Invocado después del constructor del componente, cuando la clase y sus elementos han sido inicializados. Llamado después del primer ngOnChanges
- ▶ OnChanges: Se dispara cuando se cambian el valor de un input. El método que implementamos recibe un objeto de tipo SimpleChanges con los valores
- ▶ OnDestroy: Invocado justo antes de que Angular2 destruya el componente. destruya el componente
- ▶ OnCheck: Invocado cuando el detector de cambios es ejecutado. Sirve para implementar nuestro propio handler para manejar la detección de cambios



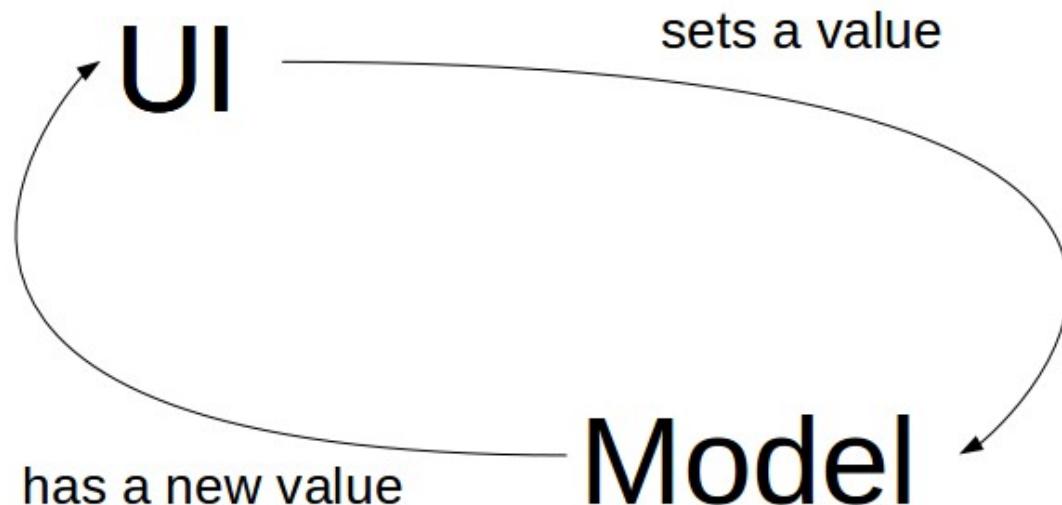


Data binding



Data Binding: Definición

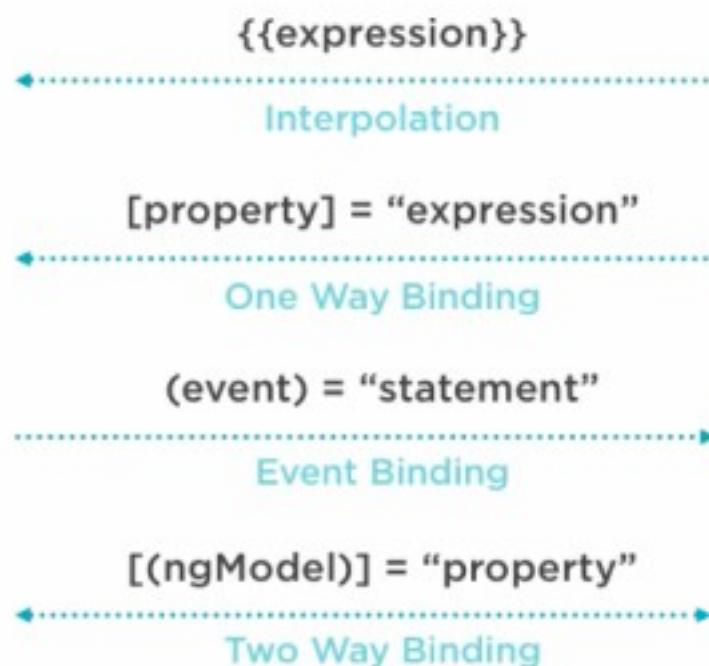
- ▶ Data binding es el proceso por el que se establece **una conexión** entre una **UI** y una **lógica de negocio**
 - ▶ Aplicado al caso de angular: conexión entre vista y componente / service
- ▶ Cuando los datos de nuestro componente cambian, los cambios se reflejan **automágicamente** en la vista



Data Binding en Angular2



DOM



Component

Data Binding: Interpolación

- ▶ Una interpolación es una **operación de lectura**
- ▶ Desde una vista, solicitamos el valor de una variable de nuestro componente
- ▶ Se utiliza la siguiente expresión:

```
<h1>{{variable_del_componente}}</h1>
```

```
3  @Component({
4    selector: 'app-root',
5    templateUrl: './app.component.html',
6    styleUrls: ['./app.component.css']
7  })
8  export class AppComponent {
9    title = 'app works!';
10 }
```

app.component.ts

```
1  <h1>
2    {{title}}
3  </h1>
4
```

app.component.html

Data Binding: one-way data binding

- ▶ One way data binding es una **operación de lectura**
- ▶ Desde una vista, solicitamos el valor de una variable de nuestro componente
- ▶ Lo mismo que el caso anterior, pero en este caso **asignamos un valor a una variable**
- ▶ Se utiliza la siguiente expresión:

```
<p [atributo] = "variable_del_documento">...</p>
```

```
<h1 [class] = "elementH1class">
  {{title}}
</h1>
```

```
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'app works!';
  elementH1class= 'title';
}
```

```
...
<h1 _ngcontent-lit-0 ng-reflect-class-name="title" class="title">
  app works!
</h1>
```

Data Binding two-way data binding

- ▶ Two-way binding es una **operación de lectura y escritura**
- ▶ La vista solicita el valor de una variable al componente (**lectura**). Desde la vista puede modificarse el valor de esta variable (**escritura**).
- ▶ Se utiliza la siguiente expresión:

```
<input [(ngModel)]="username">
```

app.component.ts

```
S
<input [(ngModel)]="username">

<p>Hello {{username}}!</p>
|
```



app.component.html

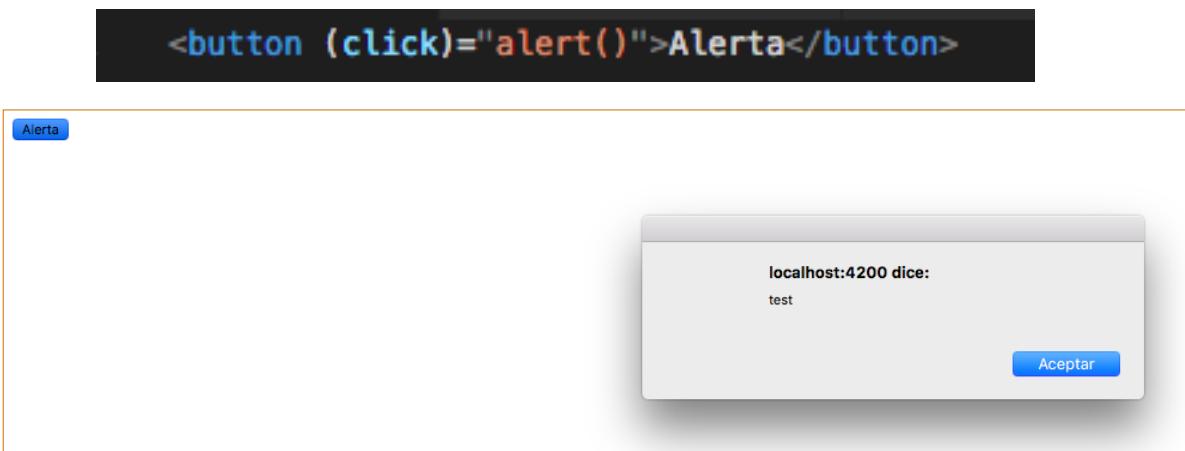
```
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  username = 'Pedro';
}
```

Pedro!!!

Hello Pedro!!!!

Data Binding event binding

- ▶ Un event binding es una **operación de escritura**
- ▶ Escribimos información en nuestro componente tras producirse un evento en nuestro DOM
- ▶ Se utiliza la siguiente expresión:
`<button (click)="sendForm()">Send</button>`
- ▶ Por defecto, angular proporciona los eventos típicos del DOM: http://www.w3schools.com/jsref/dom_obj_event.asp (sin el prefijo on y en minúsculas)



```
<button (click)="alert()>Alerta</button>

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

  alert() {
    window.alert('test');
  }
}
```



Directivas de atributo



Directivas de atributo

- ▶ Alteran la **apariencia** o **comportamiento** de un elemento DOM
- ▶ Se representan como atributos de un tag HTML
- ▶ Para crear una directiva de atributo, se utilizará el decorador @Directive
- ▶ Algunas de las directivas de atributo más típicas:
 - ▶ ngClass: Permite añadir clases CSS a un elemento HTML de forma dinámica
 - ▶ ngStyle: Al igual que ngClass, permite aplicar estilos a un elemento HTML de forma dinámica. La diferencia es que en ngStyle *hardcodeamos* directamente el estilo CSS, mientras que en ngClass indicamos clases CSS declaradas previamente
 - ▶ ngModel: Implemente un mecanismo de binding bidireccional. Refleja el valor de la variable y a su vez permite realizar cambios en esta variable (típico en elementos <input>, <select>, <button>)

Directivas de atributo : Ejemplo

<http://plnkr.co/edit/4luHtF0dzOl7t8uo0t4k?p=preview>

Directivas de atributo

- ▶ Crear una clase typescript con el decorador @Directive
 - ▶ Indicaremos el nombre del selector mediante el atributo selector
- ▶ Añadiremos la directiva creada en el atributo declarations de nuestro módulo
- ▶ Si queremos crear la directiva de forma automática: ng g directive myDirective
- ▶ Posteriormente, injectaremos el servicio ElementRef en el constructor de la clase creada
 - ▶ Este servicio nos permite manipular el DOM directamente desde la clase tipo directiva

```
import { Directive, ElementRef } from '@angular/core';
@Directive({
  selector: '[appMyDirective]'
})
export class MyDirectiveDirective {

  private elemRef: ElementRef
  constructor(elemRef : ElementRef) {
    this.elemRef = elemRef;
    this.elemRef.nativeElement.style.backgroundColor = 'blue';
  }
}
```

Directivas de atributo: Añadiendo funcionalidad

- ▶ ¿Cómo hacer más dinámicas las directivas de atributo?
- ▶ Mediante el decorador @HostListener
 - ▶ El decorador @HostListener nos permite subscribirnos a eventos DOM que ocurren en el elemento al que pertenece la directiva actual. Por ejemplo:

▶ mouseenter	scroll
▶mouseleave	dblclick
▶mousedown	...
 - ▶ Para utilizarlo, anotaremos un método de nuestra clase con el decorador @HostListener('nombre_evento_dom')

```
@HostListener('mouseenter') onMouseEnter() {
  this.elemRef.nativeElement.style.backgroundColor = 'red';
}

@HostListener('mouseleave') onMouseLeave() {
  this.elemRef.nativeElement.style.backgroundColor = null;
}
```

Ejercicio

- ▶ Vamos a crear una directiva que cambie el color de fondo de un elemento a amarillo mientras que el ratón esté situado sobre él.
 - 1. Crear un nuevo proyecto con el nombre **E04-custom-directive**
 - 2. Crear una directiva `HighlightText: ng g directive HighLightText`
 - 3. Crear los métodos y decoradores necesarios para que el color del fondo del elemento cambie al entrar el ratón
- ▶ NOTA: Para cambiar el color de fondo `-> this.el.nativeElement.style.backgroundColor = "blue";`



Directivas estructurales

Directivas estructurales

- ▶ Este tipo de directivas cambia el layout DOM añadiendo, reemplazando o eliminando elementos en el árbol.
- ▶ Algunas de las directivas estructurales más populares en Angular2:
 - ▶ `ngIf` # muestra un determinado contenido si se cumple una condición
 - ▶ `ngFor` # usado para mostrar listas de elementos
 - ▶ `ngSwitch` / `ngSwitchCase` # similar a un switch de programación.
- ▶ Los siguientes fragmentos de código muestran un ejemplo de funcionamiento de estas directivas:
 - ▶ Muestra el contenido del atributo person únicamente si el atributo person existe:

```
<div *ngIf="person">{{person}}</div>
```
 - ▶ Muestra un div por cada persona contenida en el atributo de tipo array

```
<div *ngFor="let p of people">{{p}}</div>
```

Directivas estructurales : ngSwitch

- ▶ [ngSwitch]: evalúa la expresión
- ▶ *ngSwitchCase: contiene un 'caso' para cada expresión
- ▶ *ngSwitchCaseDefault: contiene un caso por defecto. Se evalúa a true cuando ninguna de las anteriores expresiones han sido satisfechas

```
<ul [ngSwitch]="Person">
  <li *ngSwitchCase="'cat'">Hello Mohan</li>
  <li *ngSwitchCase="'Sohan'">Hello Sohan</li>
  <li *ngSwitchCase="'Vijay'">Hello Vijay</li>
  <li *ngSwitchDefault>Bye Bye</li>
</ul>
```

Ejercicio

- ▶ Crear el proyecto **E05-structural-directives**.
- ▶ Obtener una aplicación con el aspecto similar al de la siguiente imagen (ver siguiente slide):

Mostrar / Ocultar Agenda

Ver / Eliminar contactos existentes

Nombre	Apellido	Teléfono	Borrar
Pepe	García	876696954	Eliminar
Lorena	Rivera	595943944	Eliminar
Ana	Lorenzo	493848384	Eliminar
Jorge	Sanchez	523423462	Eliminar
Mario	Suarez	465645745	Eliminar
Jaime	Fernández	463463456	Eliminar
Alberto	Ruiz	634574576	Eliminar

Buscar teléfono y eliminar

Añadir nuevo contacto

Nombre Apellido Teléfono

Ejercicio

- ▶ El botón Mostrar / Ocultar Agenda oculta / muestra el resto de contenido de la web.
- ▶ La tabla de contactos es una representación gráfica de un array de contactos creado en nuestro componente:
 - ▶ Necesitaremos crear una clase Contact con los siguientes atributos: name, lastname & phone
 - ▶ Posteriormente, dentro de la clase ContactList, crearemos un atributo del tipo Contact[] y lo inicializaremos en el constructor con una serie de valores.
 - ▶ Realizaremos las tareas necesarios para que la lista de contactos quede reflejada en la tabla
- ▶ El botón Buscar teléfono y eliminar lee el valor introducido en el campo de su izquierda, busca el teléfono en el array de contactos y si lo encuentra elimina el contacto

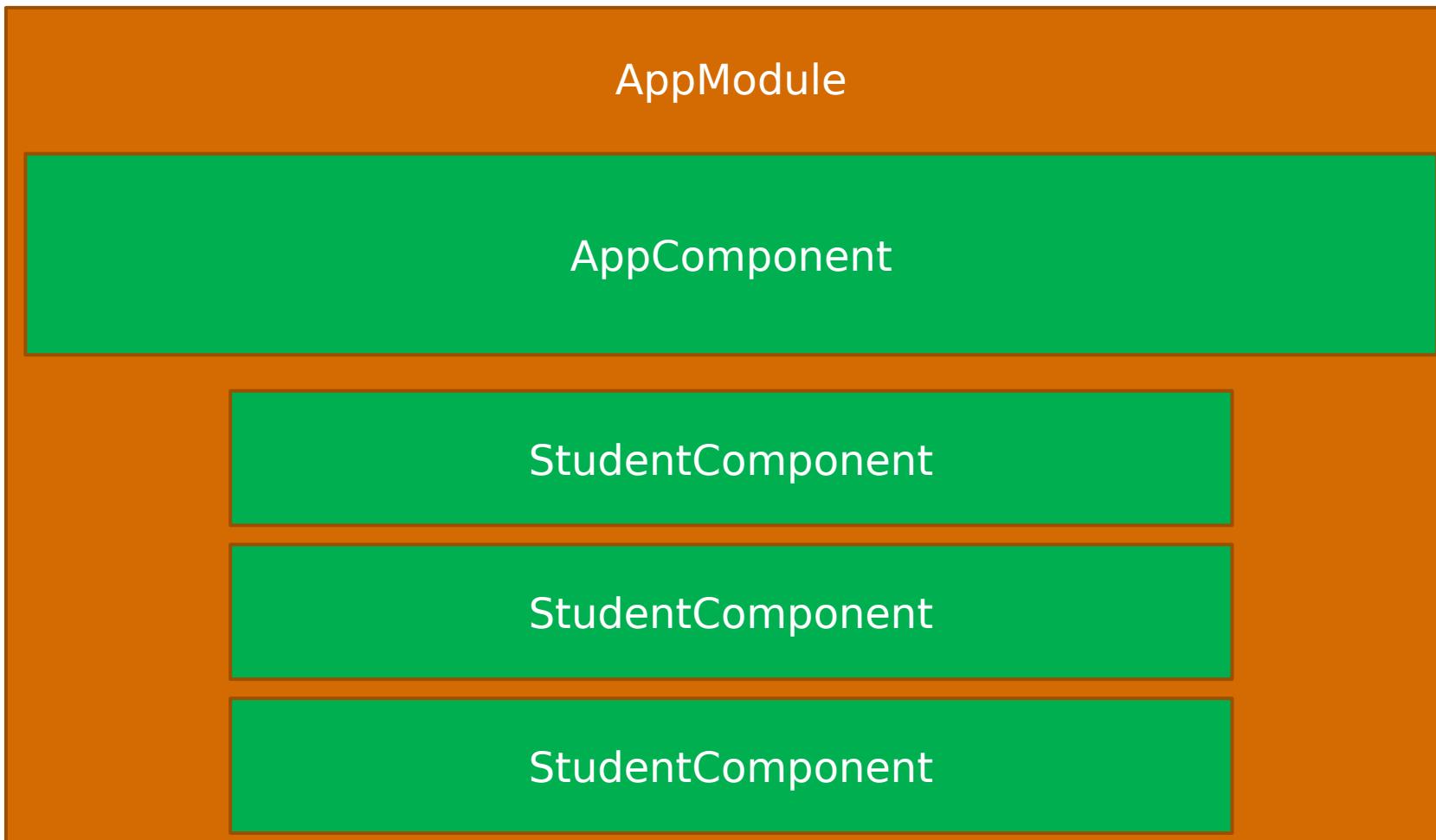
```
export class ContactListComponent implements OnInit {  
  
    private contacts: Contact[] = new Array<Contact>();  
  
    constructor() {  
        this.contacts.push(new Contact("Pepe", "García", 876696954));  
        this.contacts.push(new Contact("Lorena", "Rivera", 595943944));  
        this.contacts.push(new Contact("Ana", "Lorenzo", 493848384));  
        this.contacts.push(new Contact("Jorge", "Sanchez", 523423462));  
        this.contacts.push(new Contact("Mario", "Suarez", 465645745));  
        this.contacts.push(new Contact("Jaime", "Fernández", 463463456));  
        this.contacts.push(new Contact("Alberto", "Ruiz", 634574576));  
    }  
}
```

Ejercicio

- ▶ Crear un nuevo proyecto: **ComponentInputOutput**
- ▶ Crear una clase Student que tenga los campos: nombre y edad
 - ▶ Preparar la clase para que los atributos puedan inicializarse por el constructor
- ▶ Dentro del componente principal (AppComponent), incluiremos como atributo un array de estudiantes
 - ▶ Inicializaremos este array con 4 usuarios, creados con nombres y edades predefinidos
- ▶ En la vista del AppComponent, incluiremos la siguiente sentencia:

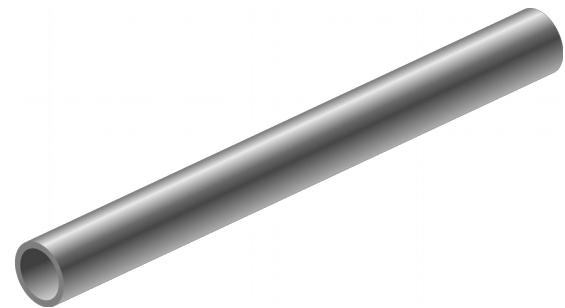
```
<app-student *ngFor=" let student of students"></app-student>
```
- ▶ Realizaremos todas las acciones necesarias para pasar el estudiante al componente hijo StudentComponent, el cual mostrará los detalles del estudiante
- ▶ Al lado de cada estudiante, tendremos un botón eliminar
 - ▶ Este botón lo único que hará será mandar un mensaje al componente padre para notificarle que hemos eliminado un usuario

Ejercicio





Pipes



Pipes

- ▶ Son funciones o filtros destinadas a transformar los datos
- ▶ Se utilizan principalmente **en las interpolaciones**
- ▶ Las pipes que Angular2 trae por defecto son: DatePipe, UpperCasePipe, LowerCasePipe, CurrencyPipe, y PercentPipe
- ▶ Se pueden concatenar varias pipes (chaining)
- ▶ Angular2 permite crear pipes personalizados
 - ng g pipe MyNewPipe
- ▶ En general, utilizaremos pipes cuando queramos mostrar información cuyo formato deba de ser editado
 - ▶ Un error muy común es crear servicios que modifiquen el valor original de la variable
 - ▶ Con las pipes, el valor original queda intacto y obtenemos código mucho más limpio: el código del componente queda intacto y solo manipulamos las vistas

```
<!-- pipesTemplate.html -->
<h1>Dates</h1>
<!-- Sep 1, 2015 -->
<p>{{date | date:'mediumDate'}}</p>
<!-- September 1, 2015 -->
<p>{{date | date:'yMMMMd'}}</p>
<!-- 3:50 pm -->
<p>{{date | date:'shortTime'}}</p>
```

```
<p>{{ birthday | date:'fullDate' | uppercase}}</p>
```

Pipes: Anatomía

- Debemos crear una clase que implemente la interfaz PipeTransform de Angular2
- Posteriormente, le añadiremos el decorador @Pipe con el parámetro name
- Implementaremos el método transform(...) de la interfaz con la lógica deseada
- Incluiremos la pipe en una interpolación de nuestra aplicación

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

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

  constructor() {}

}
```

app.component.ts

```
import { Pipe, PipeTransform } from '@angular/core';

@Pipe({
  name: 'mypipe'
})
export class MypipePipe implements PipeTransform {

  transform(value: any, toRemove: string, toAdd: string): any {
    return value.replace(new RegExp(toRemove, 'g'), toAdd);
  }
}
```

mypipe.pipe.ts

```
<p>{{title | mypipe: "a":"o"}}</p>
```

app.component.html

Pipes: Ejemplos

```
<h2>Pipe Example</h2>
<h4>1. Today is {{today}}</h4>
<h4>2. Today is {{today | date}}</h4>
<h4>3. Today is {{today | date:"dd/MM/yyyy"}}</h4>
<!-- -->
<!-- -->
<!-- -->

<h2>Decimal Pipe Example</h2>
<p>pi (no formatting): {{pi}}</p>
<p>pi (.5-5): {{pi | number:'5.5-5'}}</p>
<p>pi (2.10-10): {{pi | number:'2.10-10'}}</p>
<p>pi (.3-3): {{pi | number:'3.3-3'}}</p>
<!-- -->
<!-- -->
<!-- -->

<h2>Currency Pipe Example</h2>
<p>A in USD: {{a | currency:'USD':true}}</p>
<p>B in INR: {{b | currency:'INR':false:'4.2-2'}}</p>
```

```
<h2>Lower and Upper case Pipe Example</h2>
<p>In lowerCase : {{str | lowercase}}</p>
<p>In uppercase : {{str | uppercase}}</p>
<!-- -->
<!-- -->
<!-- -->

<h2>Percent Pipe Example</h2>
<p>myNum : {{myNum | percent}}</p>
<p>myNum (3.2-2) : {{myNum | percent:'3.2-2'}}</p>
<!-- -->
<!-- -->
<!-- -->

<h2>Slice Pipe Example</h2>
<p>{{str}} (0:4): {{str | slice:0:4}}</p>
<h4>names (1:4)</h4>
```

Ejercicio

1. Crear un nuevo proyecto **E02-custom-component**
2. Crear un nuevo componente “date” que muestre la fecha actual.
3. Cambiar el formato de la fecha al siguiente estilo¹: **Jan 3, 2017, 3:34:43 PM**
4. Cambia el estilo del reloj a tu gusto
5. Añadir un botón que actualice la fecha

¹ <https://angular.io/docs/ts/latest/api/common/index/DatePipe-pipe.html#!#examples>

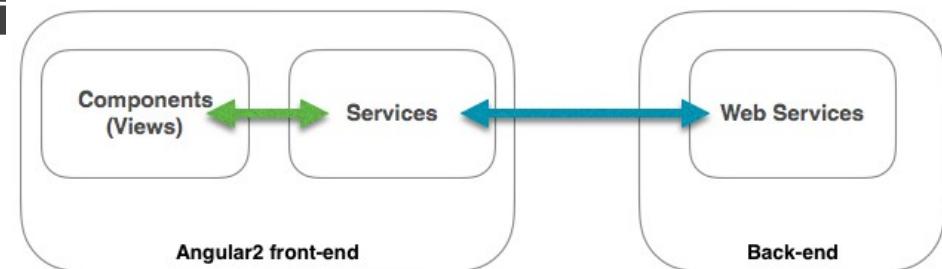


Servicios



Servicios

- ▶ Conceptualmente, los servicios en **Angular2** son **similares** a los de **AngularJS**
- ▶ Accesos a datos, variables constantes, funciones o cualquier otra característica reutilizable son **encapsuladas en servicios**
 - ▶ Lugar para encapsular la lógica de nuestra aplicación
- ▶ **Los componentes consumen servicios** específicas
 - ▶ Validación de Inputs
 - ▶ Recuperación de datos del servidor
 - ▶ Cálculos específicos
- ▶ Es necesario añadirles el decorador **@Injectable**, decorador que mantiene informado a Angular2 de que clases serán **inyectadas por dependencias**.



Servicios: inyección de dependencias

- ▶ Patrón de diseño orientado a objetos en el **que los objetos son suministrados a una clase**, en lugar de ser la propia clase la que cree dichos objetos
- ▶ Angular2 cuenta con su propio framework de inyección de dependencias
- ▶ Las clases con el decorador `@Injectable` son instanciadas automáticamente por Angular en el bootstrapping de la aplicación
- ▶ Cada servicio es instanciado una única vez por aplicación
- ▶ Deberemos indicar nuestro ~~servicio como un provider en el @NgModule~~

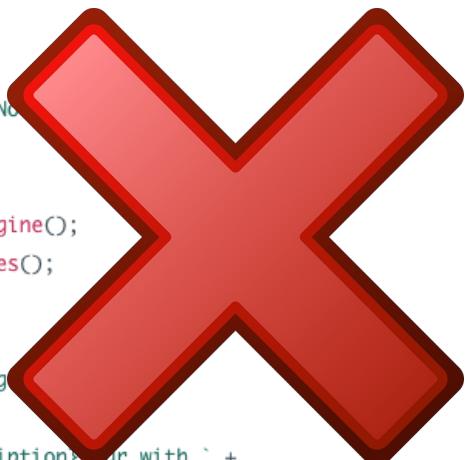
```
constructor(public engine: Engine, public tires: Tires) { }
```

```
@NgModule({
  declarations: [
    AppComponent,
    UserContainerComponent
  ],
  imports: [
    BrowserModule,
    FormsModule,
    HttpModule
  ],
  providers: [UserService],
  bootstrap: [AppComponent]
})
```

```
export class Car {
  public engine: Engine;
  public tires: Tires;
  public description = 'No car';

  constructor() {
    this.engine = new Engine();
    this.tires = new Tires();
  }

  // Method using the engine
  drive() {
    return `${this.description} car with ` +
      `${this.engine.cylinders} cylinders and ${this.tires.make} tires.`;
  }
}
```



Servicio: creación

1. Crear una clase en TypeScript con la anotación `@Injectable`.
 - ▶ Podemos usar el siguiente comando de angular-cli: `ng g service MyService`
2. Añadir el servicio creado como provider en el módulo de nuestra aplicación
3. Crear un componente desde el que utilizaremos nuestro servicio
4. Pasarlo por el constructor de nuestro componente
 - ▶ **NOTA:** No se debe inicializar el servicio directamente en el constructor, sino que debe ser pasado por el constructor.

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

  private service: UserService;
  private users: User[];

  constructor(userService: UserService) {
    this.service = userService;
  }

}
```

Ejercicio

- ▶ Duplica el proyecto **E05-structural-directives** y renómbralo a **E06-services**:
 - ▶ Actualmente, la distribución de nuestro código es bastante mala...tenemos lógica de negocio dentro de los componentes
 - ▶ Reestructura el ejercicio con el uso de servicios (solo es necesario crear uno).

Servicios: HTTP

- ▶ Angular2 incorpora por defecto un módulo para manejar peticiones HTTP
 - ▶ Se trata del HttpModule
 - ▶ Dentro de este módulo, tenemos un servicio inyectable Http que nos permite gestionar y realizar peticiones GET, POST, PUT, etc.
- ▶ Podemos usar el servicio Http desde cualquiera de nuestros servicios, inyectándolo directamente en el constructor e importándolo como provider en nuestro módulo

```
getHeroes() {
  this.heroService.getHeroes()
    .subscribe(
      heroes => this.heroes = heroes,
      error => this.errorMessage = <any>error);
}
```

```
@Injectable()
export class HeroService {
  private heroesUrl = 'app/heroes'; // URL to web API

  constructor (private http: Http) {}

  getHeroes (): Observable<Hero[]> {
    return this.http.get(this.heroesUrl)
      .map(this.extractData)
      .catch(this.handleError);
  }

  addHero (name: string): Observable<Hero> {
    let headers = new Headers({ 'Content-Type': 'application/json' });
    let options = new RequestOptions({ headers: headers });

    return this.http.post(this.heroesUrl, { name }, options)
      .map(this.extractData)
      .catch(this.handleError);
  }

  private extractData(res: Response) {
    let body = res.json();
    return body.data || {};
  }

  private handleError (error: Response | any) {
    // In a real world app, we might use a remote logging infrastructure
    let errMsg: string;
    if (error instanceof Response) {
      const body = error.json() || '';
      const err = body.error || JSON.stringify(body);
      errMsg = `${error.status} - ${error.statusText || ''} ${err}`;
    } else {
      errMsg = error.message ? error.message : error.toString();
    }
    console.error(errMsg);
    return Observable.throw(errMsg);
  }
}
```

Ejercicio

- ▶ Crear un proyecto con el nombre **E07-user-http-service**
 - ▶ Crear un servicio que, a partir de la url <https://randomuser.me/api/?results=5>, muestre los datos de los usuarios
 - ▶ Crear un componente que almacene un array de Usuarios (recomendable crear una clase User)
 - ▶ Invocar el servicio desde nuestro componente y mostrar los atributos para cada uno de los usuarios recibidos

```
firstname : string;  
lastname : string;  
email: string;  
imgUrl: string;  
phone: string;
```



Rutas



Rutas

- ▶ Nos permiten definir **rutas en el lado del cliente**
- ▶ Cada ruta es enlazada a un componente (ruta -> comportamiento específico)
- ▶ Las rutas, generalmente, se definen **a nivel de módulo**
- ▶ Una ruta cuenta con los siguientes parámetros:
 - ▶ **path**: Path de la URL. El valor '' hace referencia al path por defecto. El valor ** enlaza a todas aquellas rutas que no hayan sido definidas.
 - ▶ **component**: Componente asignado a la ruta actual
 - ▶ **redirectTo (opcional)**: Redirecciona a otra ruta **previamente definida**
 - ▶ **pathMatch (opcional)**: Requerido si se usa redirectTo. Valores válidos 'prefix' o 'full'
 - ▶ **data (opcional)**: Datos arbitrarios que pasamos al componente cuando se ejecuta nuestra ruta
- ▶ Una ruta admite parámetros **dinámicos** y parámetros **estáticos**

Rutas: parámetros dinámicos

- ▶ Podemos definir rutas con parámetros <localhost:4200/heroes/3>
- ▶ Se indicarán directamente en el path con el formato :<ruta>/:param
- ▶ Para acceder a este parámetro desde el componente HeroDetailComponent, inyectaremos el servicio ActivatedRoute de Angular2 en el constructor
 - ▶ `this.route.snapshot.params['<nombre_parámetro>']`
- ▶ Para acceder a las rutas desde enlaces HTML (`<a>`), utilizaremos el atributo routerLink proporcionado por Angular2 (en lugar del clásico href)

```
constructor(  
    private route: ActivatedRoute,  
    private router: Router,  
    private service: HeroService  
) {}
```

```
<!-- Actualmente estamos en la URL : /heroes -->  
<a routerLink=".//{hero.id}">{{hero.name}}</a>
```

```
const appRoutes: Routes = [  
  { path: 'crisis-center', component: CrisisListComponent },  
  { path: 'hero/:id', component: HeroDetailComponent },  
  {  
    path: 'heroes',  
    component: HeroListComponent,  
    data: { title: 'Heroes List' }  
  },  
  { path: '',  
    redirectTo: '/heroes',  
    pathMatch: 'full'  
  },  
  { path: '**', component: PageNotFoundComponent }  
];
```

Rutas: parámetros estáticos

- ▶ También puede ser interesante enviar **datos estáticos para cada una de las rutas**:
 - ▶ Nombre de la ruta para situar en la cabecera, color de fondo, etc.
 - ▶ Utilizaremos el objeto data **dentro de la definición de cada ruta**
 - ▶ En este objeto podremos incluir tantos datos como queramos.
 - ▶ Para recogerlos desde nuestro componente, utilizaremos la propiedad data del atributo “snapshot” del servicio **ActivatedRoute**

```
{  
  path: 'test',  
  component: MessageDetailComponent,  
  data: {  
    valuePassed: 'test'  
  }  
},
```

```
private myStaticDataValue;  
constructor(  
  private route: ActivatedRoute) {  
}  
  
ngOnInit() {  
  this.myStaticDataValue = this.route.snapshot.data['valuePassed'];  
}
```

Rutas: navegación estática

- ▶ La navegación estática entre rutas se define con el elemento `<a>...` de HTML5.
- ▶ **No se debe utilizar href para los links.** ¿Por qué?
 - ▶ Angular2 nos proporciona su propia directiva `routerLink`

```
<a routerLink="/miRuta">Ruta Definida</a>
```

- ▶ Además, cuando utilicemos rutas necesitaremos definir **donde se carga nuestro contenido**

- ▶ Para esto utilizaremos el tag `<router-outlet></router-outlet>`

```
<div class="demo-layout-transparent mdl-layout mdl-js-layout">
  <header class="mdl-layout__header mdl-layout__header--transparent">
    <div class="mdl-layout__header-row">
      <!-- Title -->
      <span class="mdl-layout-title">Scotch Pets</span>
      <!-- Add spacer, to align navigation to the right -->
      <div class="mdl-layout-spacer"></div>
      <!-- Navigation with router directives-->
      <nav class="mdl-navigation">
        <a class="mdl-navigation__link" routerLink="/">Home</a>
        <a class="mdl-navigation__link" routerLink="/cats">Cats</a>
        <a class="mdl-navigation__link" routerLink="/dogs">Dogs</a>
      </nav>
    </div>
  </header>
  <main class="mdl-layout__content">
    <h1 class="header-text">We care about pets...</h1>
  </main>
</div>
<!-- Router Outlet -->
<div class="container">
  <router-outlet></router-outlet>
</div>
```

Rutas: navegación dinámica

- ▶ Hasta ahora, la navegación por las rutas solo se producía a través de las vistas, mediante la directiva [routerLink]
- ▶ ¿Se puede navegar de forma más dinámica, por ejemplo, desde nuestro componente?
 - ▶ **Ejemplo:** queremos cambiar la vista cuando se dispare un evento en nuestro componente
- ▶ ¡Sí! Mediante el servicio Router
- ▶ Router posee dos métodos principales para la navegación:
 - ▶ **navigate([]):** El primer parámetro es la ruta a la que deseamos acceder. El segundo parámetro es opcional y contiene posibles datos estáticos. Utilizado para navegar por rutas relativas
 - ▶ **navigateByUrl():** Similar al anterior, pero con rutas absolutas

Rutas: navegación dinámica

```
export class PersonDetailsComponent implements OnInit, OnDestroy {
  person: Person;
  sub: any;

  constructor(private peopleService: PeopleService,
              private route: ActivatedRoute,
              private router: Router){
  }

  ngOnInit(){
    this.sub = this.route.params.subscribe(params => {
      let id = Number.parseInt(params['id']);
      this.person = this.peopleService.get(id);
    });
  }

  ngOnDestroy(){
    this.sub.unsubscribe();
  }

  gotoPeopleList(){
    let link = ['/persons'];
    this.router.navigate(link);
  }
}
```

Rutas: creación

1. Generar un módulo para almacenar las rutas : (buenas prácticas un módulo para definir todas las rutas) -> **ng new MyApp -routing**
 - ▶ El parámetro `-routing` crea un fichero módulo adicional (`app-routing.module.ts`) para la gestión de las rutas
2. Definir en el fichero `app-routing.module.ts` las rutas deseadas (con los parámetros explicados anteriormente)
 - ▶ Definirlas en el array precreado `const routes : Routes []`
 - ▶ Crear un controlador para cada una de las rutas definidas
 - ▶ Opcional: Insertar el servicio `ActivatedRoute` y acceder, si fuese necesario, a los parámetros mediante
`this.activatedRoute.snapshot.params['<nombre_parametro>']`

Ejercicio

1. Generar el proyecto con rutas: **E09-routes**
2. Crear una aplicación con las siguientes rutas:
 1. **/profile**: Muestra información básica del perfil (nombre, apellidos, edad). Deberá ser la ruta por defecto cuando accedemos a directorio raíz.
 2. **/messages**: Muestra una lista de mensajes
 - ▶ Crear un servicio que devuelva 3 o 4 mensajes con la estructura : título y mensaje
 - ▶ **/messages/:id**: Muestra los detalles de un mensaje concreto
 - ▶ **/logout**: Ruta simple. deberá mostrar “Usted ha salido de la aplicación”

/, /profile

/messages

/messages/1

/logout

[Perfil](#) [Mensajes](#) [Salir](#)

Bienvenido a mi perfil:

- Nombre: Nacho
- Apellidos: García Fernández
- Edad: 25

[Perfil](#) [Mensajes](#) [Salir](#)

[Saludo](#)

[Recuerdos de tu tío](#)

[Quedada el fin de semana](#)

[Perfil](#) [Mensajes](#) [Salir](#)

[Saludo](#)

[hola, qué tal estás?](#)

[Todos los mensajes](#)

[Saludo](#)

[Recuerdos de tu tío](#)

[Quedada el fin de semana](#)

[Perfil](#) [Mensajes](#) [Salir](#)

logout works!

Ejercicio

- ▶ Para cada ruta, definir y mostrar (de forma estática ;)):
 - ▶ Un título: “Mi perfil”, “Lista de mensajes”, “Mensaje”, “Salir de la aplicación”
 - ▶ Un color de fondo
- ▶ En la lista de mensajes, incorporar un botón que nos redirija al perfil del usuario

/, /profile

Perfil Mensajes Salir

Bienvenido a mi perfil:

- Nombre: Nacho
- Apellidos: García Fernández
- Edad: 25

/messages

Perfil Mensajes Salir

Saludo

Recuerdos de tu tío

Quedada el fin de semana

/messages/1

Perfil Mensajes Salir

Saludo

hola, qué tal estás?

Todos los mensajes

Saludo

Recuerdos de tu tío

Quedada el fin de semana

/logout

Perfil Mensajes Salir

logout works!

Rutas observables VS Snapshots

- ▶ Hasta ahora, hemos visto que accedemos a los parámetros del Router desde nuestro componente con la siguiente línea:

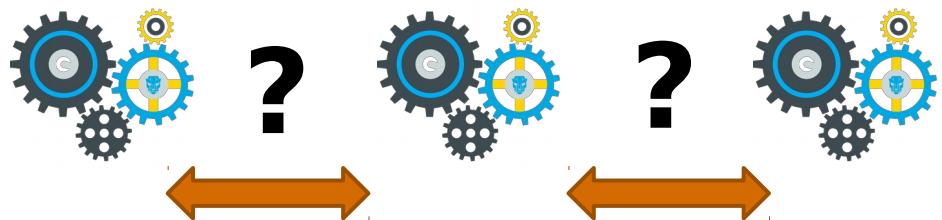
```
this.message = this.messageService.getMessage(this.route.snapshot.params['id']);
```

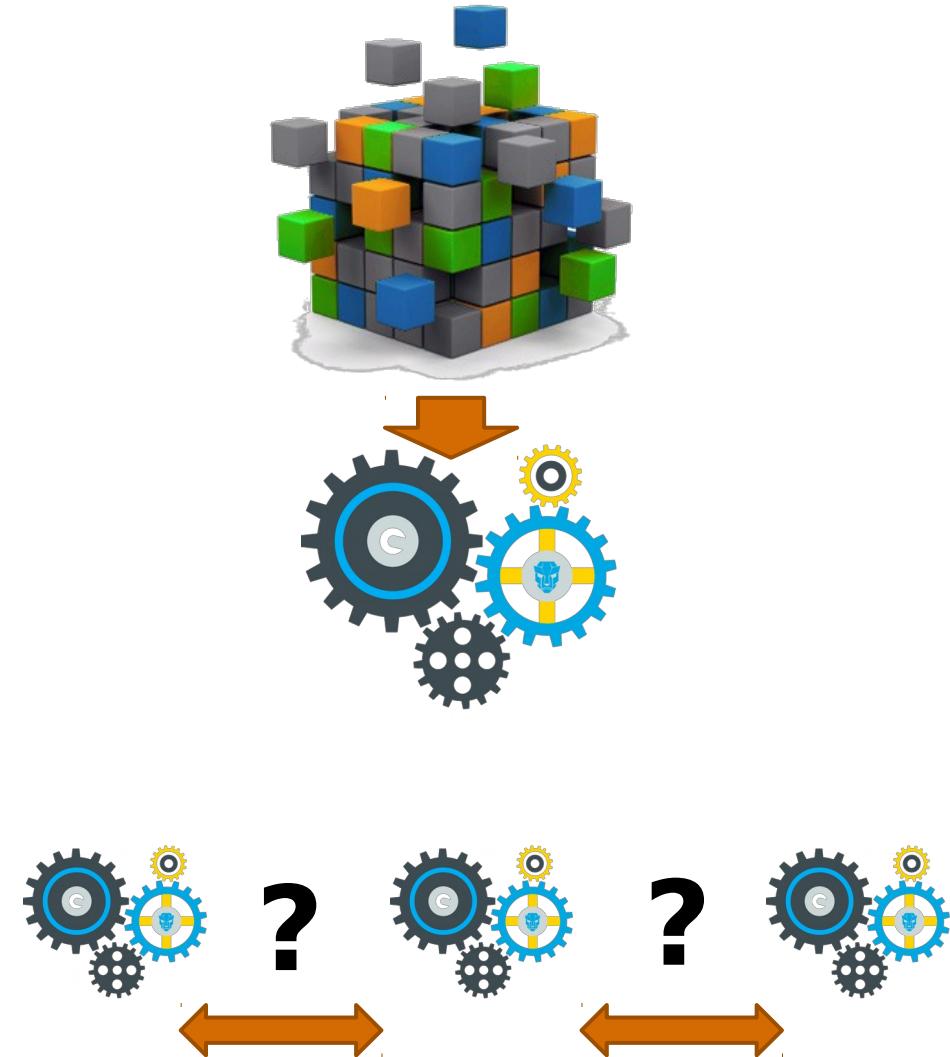
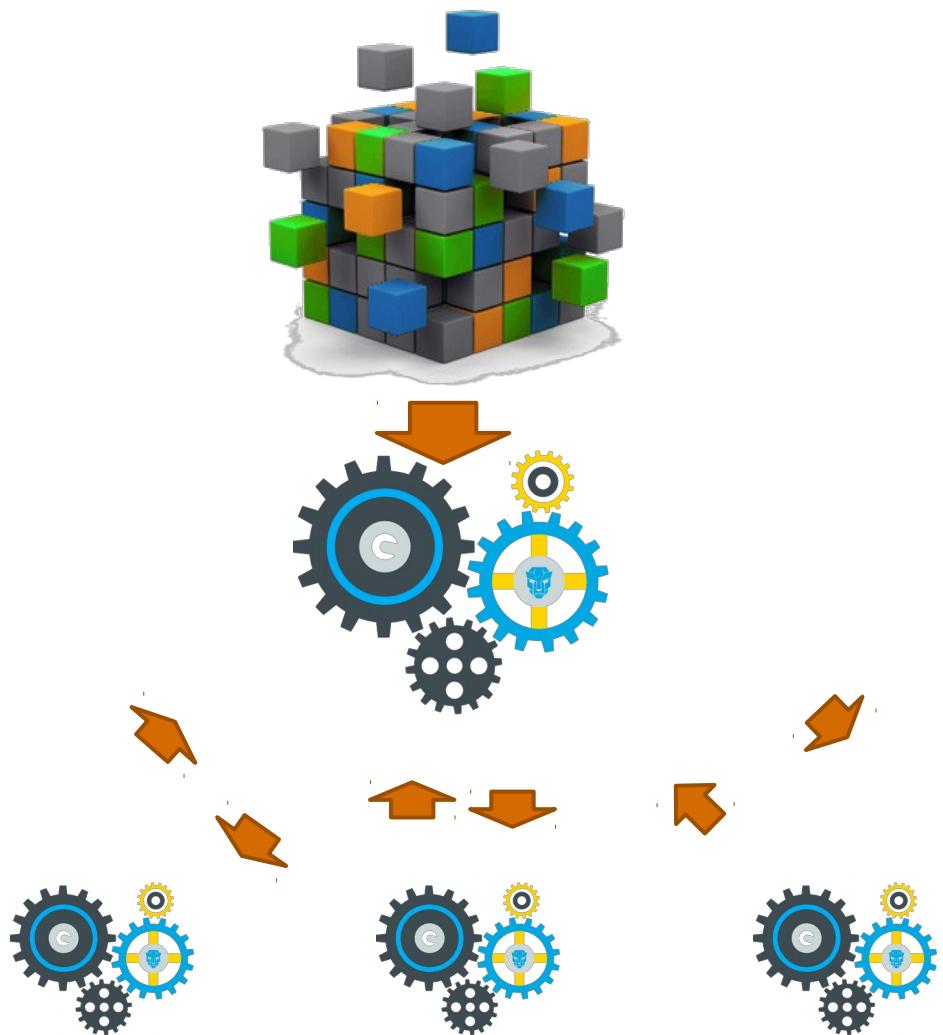
- ▶ **¿Problema?** ¡No es observable!
 - ▶ Si desde la URL del héroe actual (/hero/2) queremos navegar a otro héroe (/hero/3), los cambios no se producirán.
 - ▶ El valor del `this.route.snapshot.params['id']` seguirá siendo el primero (2).
- ▶ **¿Solución?** Observable y suscripción:
 - ▶ Simplemente sustituyendo la línea anterior con una suscripción al servicio de parámetros, solucionaremos el problema:

```
this.route.params.subscribe((params : Params) => this.message = this.messageService.getMessage(params['id']));
```



Interacción entre componentes del mismo nivel





Intercambio datos: componente - servicio - componente

- ▶ Hasta ahora hemos visto una forma sencilla de **intercambiar datos entre un componente padre y sus hijos**
 - ▶ Mediante el decorador @Input y @Output
- ▶ Pero...¿cómo puedo intercambiar datos entre **dos componentes del mismo nivel?**
 - ▶ Mediante el uso de observables / observer
 - ▶ Los observables solo pueden escuchar cambios, pero no pueden enviar datos
 - ▶ Para ello utilizaremos la clase Subject, en lugar de Observable
 - ▶ Un Subject es un observable que a su vez es un observer, por lo que puede emitir datos y escuchar por esos cambios

Subject: Ejemplo

```
// Regular Subject

let subject = new Subject();

subject.next("b");

subject.subscribe((value) => {
  console.log("Subscription got", value); // Subscription wont get
                                              // anything at this point
});

subject.next("c"); // Subscription got c
subject.next("d"); // Subscription got d
```



Repaso de conceptos



Jasmine

Jasmine

- ▶ Framework **open-source** para realizar tests en Javascript
 - ▶ No tiene dependencias con otras librerías
 - ▶ No requiere un DOM para ejecutarse
 - ▶ Sintaxis clara, simple y concisa
- ▶ Destinado al **Behavior-driven** development (BDD)
 - ▶ Variante del test-driven development (TDD)
 - ▶ Requisitos son transformados en test muy específicos y adaptamos nuestro software para pasar los tests (cíclico)
 - ▶ En BDD no probamos solo unidades o clases. **Probamos escenarios** y el **comportamiento** de las clases en estos escenarios

<https://jasmine.github.io/2.0/introduction.html>

Jasmine : Ejemplo

```
describe("A suite", function() {
  it("contains spec with an expectation", function() {
    expect(true).toBe(true);
  });
});
```

```
it("The 'toBeLessThan' matcher is for mathematical comparisons", function() {
  var pi = 3.1415926,
      e = 2.78;

  expect(e).toBeLessThan(pi);
  expect(pi).not.toBeLessThan(e);
});
```

```
it("The 'toThrowError' matcher is for testing a specific thrown exception", function() {
  var foo = function() {
    throw new TypeError("foo bar baz");
  };

  expect(foo).toThrowError("foo bar baz");
  expect(foo).toThrowError(/bar/);
  expect(foo).toThrowError(TypeError);
  expect(foo).toThrowError(TypeError, "foo bar baz");
});
```

<https://jasmine.github.io/2.0/introduction.html>

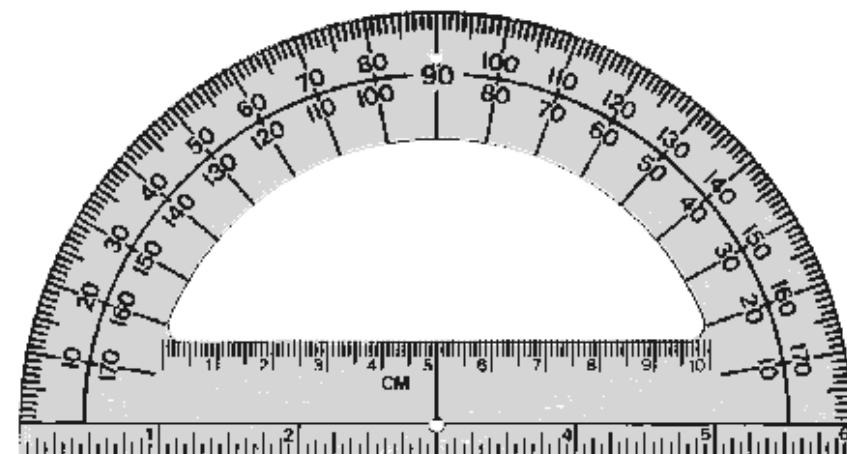


Repaso de conceptos



Protractor

- ▶ Framework de testing **extremo a extremo (e2e)** creado específicamente para AngularJS /Angular2
- ▶ Ejecuta las pruebas en un navegador real
 - ▶ Interactúa automáticamente con el navegador
- ▶ Al igual que Angular, es una herramienta **desarrollada y mantenida por Google**
- ▶ Funciona sobre Selenium
 - ▶ Framework de testing destinado a probar aplicaciones web completas



<https://jasmine.github.io/2.0/introduction.html>

Protractor : Ejemplos

```
describe('angularjs homepage', function() {
  it('should greet the named user', function() {
    browser.get('http://www.angularjs.org');

    element(by.model('yourName')).sendKeys('Julie');

    var greeting = element(by.binding('yourName'));

    expect(greeting.getText()).toEqual('Hello Julie!');
  });
});
```

```
describe('todo list', function() {
  var todoList;

  beforeEach(function() {
    browser.get('http://www.angularjs.org');

    todoList = element.all(by.repeater('todo in todoList.todos'));
  });

  it('should list todos', function() {
    expect(todoList.count()).toEqual(2);
    expect(todoList.get(1).getText()).toEqual('build an AngularJS app');
  });

  it('should add a todo', function() {
    var addTodo = element(by.model('todoList.todoText'));
    var addButton = element(by.css('[value="add"]'));

    addTodo.sendKeys('write a protractor test');
    addButton.click();

    expect(todoList.count()).toEqual(3);
    expect(todoList.get(2).getText()).toEqual('write a protractor test');
  });
});
```

Validación de test

- ▶ Crear una nueva app **testApp**
- ▶ Ejecutar los tests unitarios: `ng test`
- ▶ Ejecutar los tests e2e:
 - ▶ `ng serve`
 - ▶ `ng e2e`

