



## ANGULAR 2

© 2017, ACTIBYTI PROJECT SLU, Barcelona  
Autor: Ricardo Ahumada



MINISTERIO  
DE ENERGÍA, TURISMO  
Y AGENDA DIGITAL

red.es



ESTRATEGIA DE  
EMPRENDIMIENTO Y  
EMPLEO JUVENIL  
*garantía juvenil*



UNIÓN EUROPEA

Fondo Social Europeo  
“El FSE invierte en tu futuro”

# ÍNDICE DE CONTENIDOS

1. Introducción a Angular2
2. Componentes
3. Templates, Interpolación y Directivas
4. Data Binding y Pipes
5. Componentes Anidados
6. Extensión de Componentes
7. Rutas y Navegación
8. Creación y uso de Servicios
9. Acceso a APIs de servidor

1

# INTRODUCCIÓN

# Angular 2

- Angular 2 es una plataforma de desarrollo para construir aplicaciones web, tanto móviles como de escritorio “todo en uno”.
- Desarrollado por el equipo Angular de Google (Igor Minar, Brad Green, and Misko Hevery\*)
- Angular 2 es “cross platform”: usa las capacidades de las plataformas web modernas; permite crear aplicaciones móviles nativas y; generar aplicaciones de escritorio [instaladas].
- Se ha mejorado sus prestaciones de velocidad y rendimiento.
- Se ha optimizado sus características de productividad (sintaxis, CLI e IDEs)
- Se ha creado un proceso de desarrollo completo que incluye además el testing, animaciones y accesibilidad.

# Porqué Angular 2

- Móvil
  - › Enfocado en desarrollo de apps móviles
- Modular
  - › Mejor performance
  - › Coger las partes que sólo se necesitan
- Moderno
  - › ES6 y navegadores modernos “evergreen”

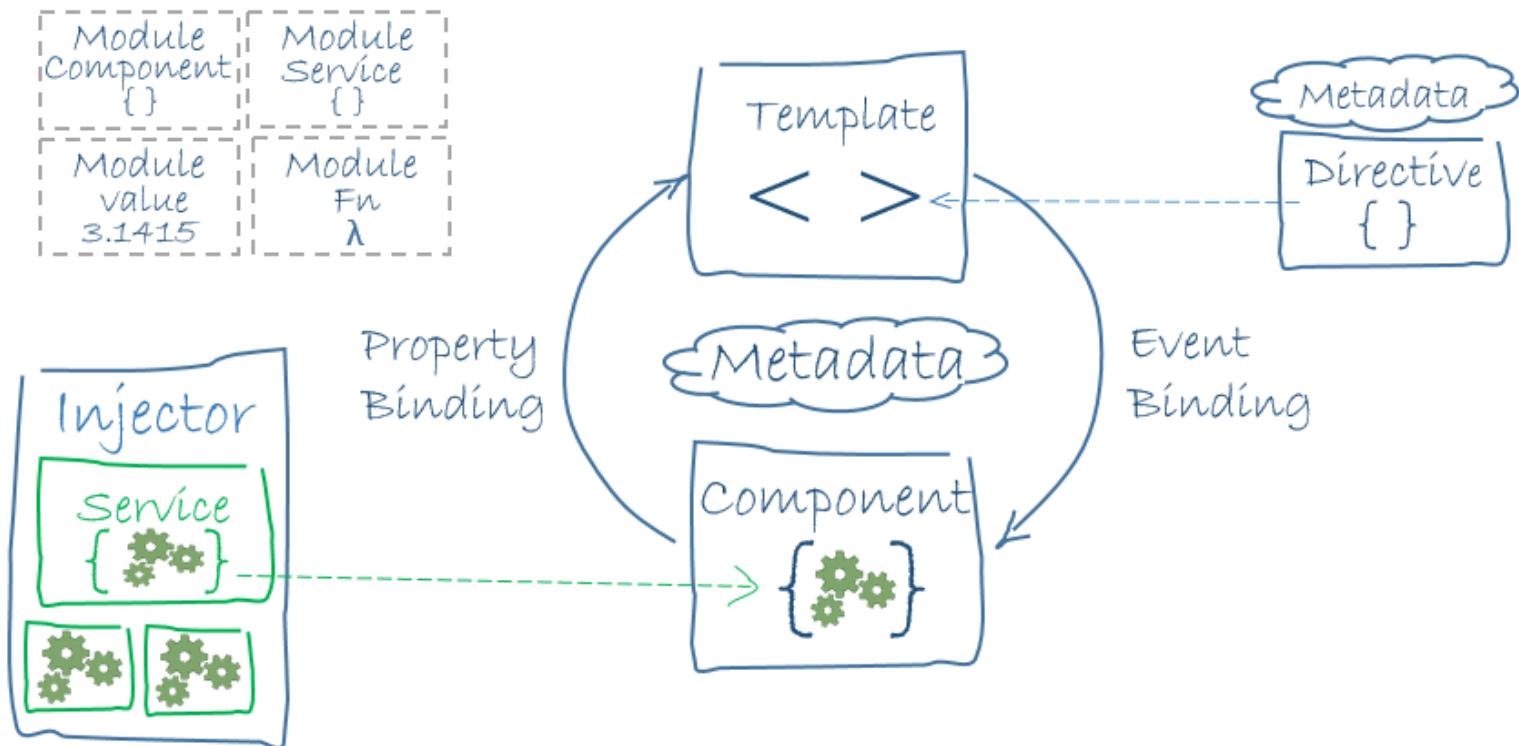
# Principales cambios respecto a Angular 1

- TypeScript
  - Inicialmente AtScript → Ahora TypeScript
  - No es obligatorio, pero sí muy recomendable (inevitable)
- Mejora en la Inyección de dependencias (DI)
  - Anotaciones
  - Injectors hijos
  - Control del lifetime/scope
- Plantillas y DataBinding
  - Carga dinámica (on the fly)
  - Compilación asíncrona de templates → carga de las dependencias simplemente referenciándolas en la definición del componente
  - Directivas de: Componente, Decorator, Template

# Filosofía básica de Angular 2

- Separación de responsabilidades (primer principio SOLID)
- Fomenta la separación de la manipulación HTML y la lógica de control.
- Separación entre el servidor y la página Web.
- Bien estructurado en varias formas:
  - El diseño de la interfaz de usuario
  - La lógica de negocio
  - Test unitarios y de uso (Jasmine, Protractor, Karma, etc.)
  - Incluso en la documentación oficial se aportan plantillas de Tests para cada demo que se muestra.

# Arquitectura Angular 2

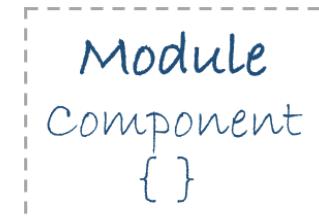


- La arquitectura Angular2 está basada en componentes que adquieren diferentes responsabilidades según se indique con los decorators.

# Arquitectura

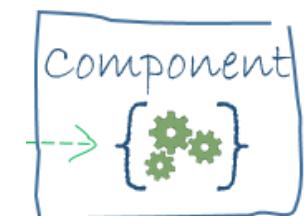
## ➤ El módulo

- › Un bloque dedicado a un solo propósito
- › Contienen y exportan (hacen públicos) componentes (clases, objetos, ...)
  - › app.component.ts → export class AppComponent {}
- › Cuando necesitamos un componente se importan
  - › Import {AppComponent} from './app.component'
- › Algunos módulos son librerías de otros módulos → façade



## ➤ El componente

- › Controla un bloque de la vista de la App
- › Dentro de una clase → interactúa con la API de la vista
- › Se crean y destruyen conforme el usuario se mueve en la app
- › Pueden tener componentes hijos

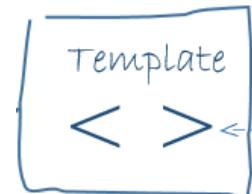


# Arquitectura

## ➤ La plantilla

- Un bloque dedicado a un solo propósito
- Sintaxis de plantilla:

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



## ➤ Los metadatos

- Se adjuntan a las clases para convertirlas en componentes
- Decorators de clase:
  - @Component: define una clase como un componente
  - @Directive: define una directiva (añade funcionalidad a un elementos DOM)
  - @Pipe: define un pipe (e.g. filtro)
  - @Injectable: Indica que la clase tiene dependencias que deben ser inyectadas en el constructor



# Arquitectura

## ➤ Los metadatos

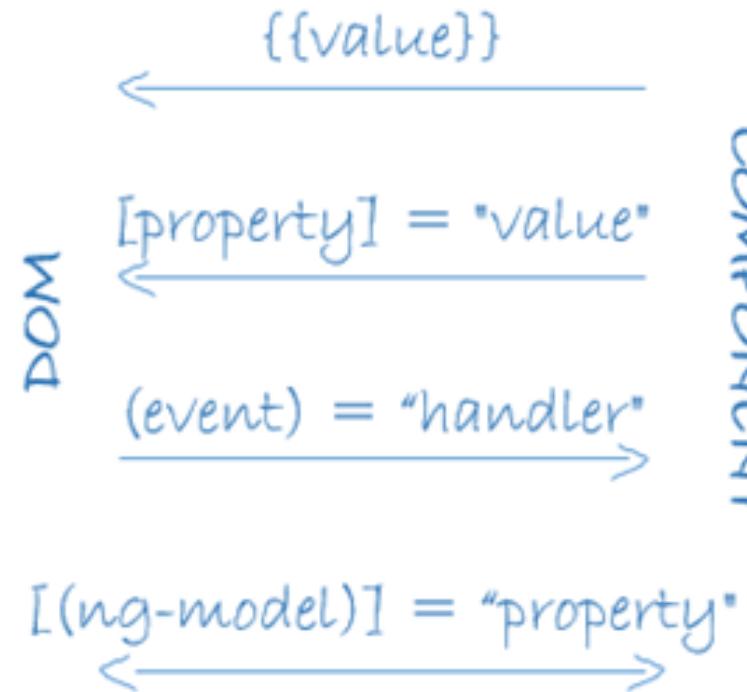


- Decorators de atributos de clase
  - @Input() myProperty: una propiedad [de entrada] que se puede actualizar mediante binding
  - @Output() myEvent = new EventEmitter(): una propiedad [de salida] que dispara un evento
  - @HostBinding('[class.valid]') isValid: Enlaza a una propiedad de host (ej. CSS)
  - @HostListener('click', ['\$event']) onClick(e) {...} : Enlaza a un evento de host (ej onclick)
  - @ContentChild(myPredicate) myChildComponent;
  - @ContentChildren(myPredicate) myChildComponents;
  - @ViewChild(myPredicate) myChildComponent;
  - @ViewChildren(myPredicate) myChildComponents;
- Decorators de enrutado
  - @RouteConfig: permite definir rutas para el componente
  - @CanActivate: Define una función que el enrutador deberá llamar primero para determinar si activar el componente

# Arquitectura

## > Data Binding

- › Permite poner datos en el html y transformar las respuestas de los usuarios en acciones y actualización de datos.
- › *Binding de componente a DOM:*
  - › {{valor}}
  - › [propiedad] = "valor"
- › *Binding de DOM a componente*
  - › (evento) = "handler"
- › *Binding de doble sentido*
  - › [(ng-model)] = "propiedad"
- › También permite enlazar datos entre componentes padres e hijos



# Arquitectura

## ➤ La directiva

- › Las plantillas son dinámicas → se renderizan de acuerdo a
- › Instrucciones de directiva
- › Es una clase con metadatos de directiva @Directive
  - › Estructurales
  - › De atributo



## ➤ El servicio

- › Valor, función, característica que necesite nuestra aplicación
- › Es una clase con un propósito específico (SR)
- › Los componentes son consumidores de servicios



# Arquitectura

## ➤ Inyección de Dependencias

- Suministra una nueva instancia de clase con todas sus dependencias
- Servicios necesarios → en función del tipo de los parámetros de constr.

Component  
Service  
{Constructor(service)}

# Plantillas de desarrollo

- Para agilizar el desarrollo de aplicaciones se usa plantillas que ya vienen configuradas con las herramientas necesarias para compilar, desplegar, minimizar/ofuscar y testar el código
- Hay varias plantillas disponibles. Destacan entre ellas:
  - Plantillas usando Webpack
  - Plantillas usando la herramienta Angular CLI

# Webpack

- Empaquetador (bundler) de módulos muy potente y bastante usado en la industria
  - Permite empaquetar los activos de una aplicación para su distribución
- Se configura a través de un archivo js: webpack.config.js
  - **entry**: define los componentes de entrada de la aplicación. Sigue el trazo de los imports.
    - resolve: las extensiones que debe resolver para los imports cuando no los tenga
  - **output**: el fichero de empaquetado que resultará del proceso
  - **loaders**: webpack puede empaquetar cualquier tipo de fichero: JavaScript, TypeScript, CSS, SASS, LESS, images, html, fonts,...lo hace a través de la configuración de los loaders, por cada tipo de fichero.
  - **plugins**: permite introducir pasos personalizados en el proceso de empaquetado (como la minificación o la ofuscación del código)



# Plantillas base: Webpack template



# Clonar la plantilla

## ➤ Instalar webpack:

- › npm install -g webpack

## ➤ Servidor webpack:

- › npm install webpack-dev-server -g

## ➤ Clonar la plantilla basada en webpack

- › git <https://github.com/ricardoahumada/angular-webpack.git>

## ➤ Inicializa el servidor

- › npm start
- › Esto inicializará el servidor en el puerto: 7770
- › Accede en tu navegador en la dirección: <http://localhost:7770/>

## ➤ Para el servidor

- › En la ventana de consola: Ctrl+c
- › Escribe “s” y pulsa “enter”



# Estructura de la plantilla

- **src/app**: directorio con los módulos y componentes la aplicación
  - index.html: archivo de entrada de la aplicación
  - main.ts: clase carga de la aplicación
- **package.json**: configuración del proyecto y de dependencias
- **webpack.config.js**: configuración para empaquetado → config/webpack.dev.js
- **tsconfig.json**: configuración de ts
- **karma.conf.js**: configuración para karma (gestor de testing) → config/karma.conf.js
- **config**: directorio con los detalles de configuración
- **public**: directorio con elementos estáticos

```
▼ config
    helpers.js
    karma-test-shim.js
    karma.conf.js
    webpack.common.js
    webpack.dev.js
    webpack.prod.js
    webpack.test.js
▶ dist
▶ node_modules
▶ public
▼ src
▶ app
    index.html
    main.ts
    polyfills.ts
    vendor.ts
    karma.conf.js
    package.json
    tsconfig.json
    webpack.config.js
```

# Angular cli

- Es un cliente que permite generar prototipos de manera interactiva y muy sencilla
- Algunas órdenes comunes:
  - *ng new <proyecto>*: crea un nuevo proyecto
  - *ng serve*: inicia el servidor
  - *ng build [-prod/-dev -e=prod/dev]*: genera el empaquetado
  - *ng g <tipo\_de\_componente>*: genera un componente de alguno de los tipos siguientes:

Componente	<b>ng g componente my-new-component</b>
Directiva	<i>ng g directive my-new-directive</i>
Pipe/filtro	<i>ng g pipe my-new-pipe</i>
Servicio	<i>ng g service my-new-service</i>
Clase	<i>ng g class my-new-class</i>
Interface	<i>ng g interface my-new-interface</i>
Enum	<i>ng g enum my-new-enum</i>
Módulo	<i>ng g module my-module</i>



# Plantillas base: Angular Cli



# Generar la plantilla

- **Instala angular cli**
  - `npm install -g @angular/cli`
- Documentación: <https://github.com/angular/angular-cli>
- Crea un proyecto:
  - `ng new my-new-app`
  - `cd my-new-app`
- **Iniciar el servidor**
  - `ng serve`
- **Parar el servidor**
  - En la ventana de consola: `Ctrl+c`
  - Escribe “s” y pulsa “enter”



# Estructura de la plantilla

- **src/app**: directorio con los módulos y componentes la aplicación
  - *main.ts*: clase de carga de la aplicación
  - *tsconfig.json*: configuración de ts
- **index.html**: archivo de entrada de la aplicación
- **package.json**: configuración del proyecto y de dependencias
- **angular-cli.json**: configuración del cli para empaquetar, testar, etc..
- **karma.conf.js**: configuración para karma (gestor de testing)
- **tslint.json**: configuración del linter de código ts (calidad de código)
- **protractor.config.js**: configuración de protractor (tests end to end o cómo los usuario ven la app)
- **e2e**: directorio con las especificaciones de los tests para protractor
- **src/environments**: directorio con la configuración de entornos

```
▶ └── dist
▶ └── e2e
▶ └── node_modules
└── src
    ├── app
    ├── assets
    ├── environments
    ├── favicon.ico
    ├── index.html
    ├── main.ts
    ├── polyfills.ts
    ├── styles.css
    ├── test.ts
    ├── tsconfig.json
    └── .editorconfig
        └── .gitignore
            ├── angular-cli.json
            ├── karma.conf.js
            ├── package.json
            ├── protractor.conf.js
            ├── README.md
            └── tslint.json
```



## Pongámoslo en práctica

- Crea El esqueleto de la app TareasApp , usando la plantilla webpack o AngularCLI
- Por ejemplo, clonaremos la plantilla webpack y le asignaremos un nombre
  - git clone https://github.com/ricardoahumada/angular-webpack.git TareasApp
  - cd TareasApp
  - npm install
  - npm start

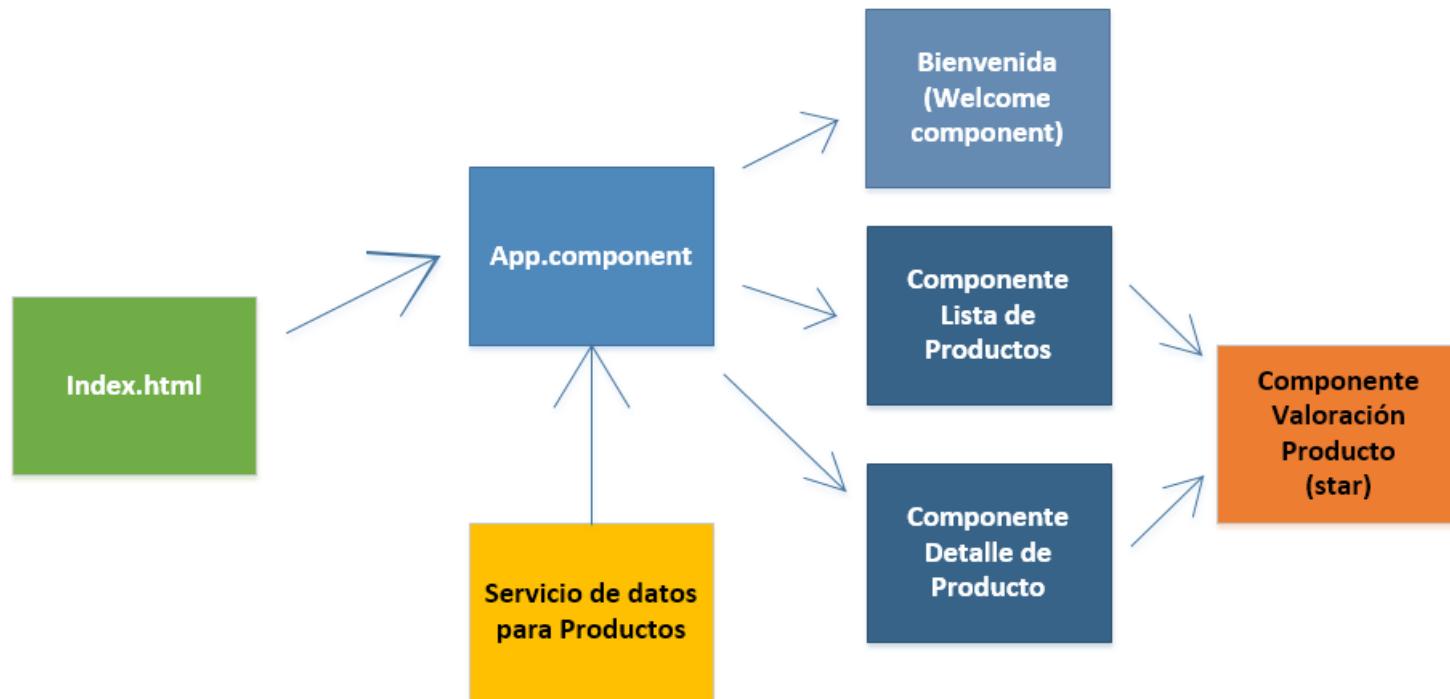
3

# COMPONENTES

# Construcción de componentes

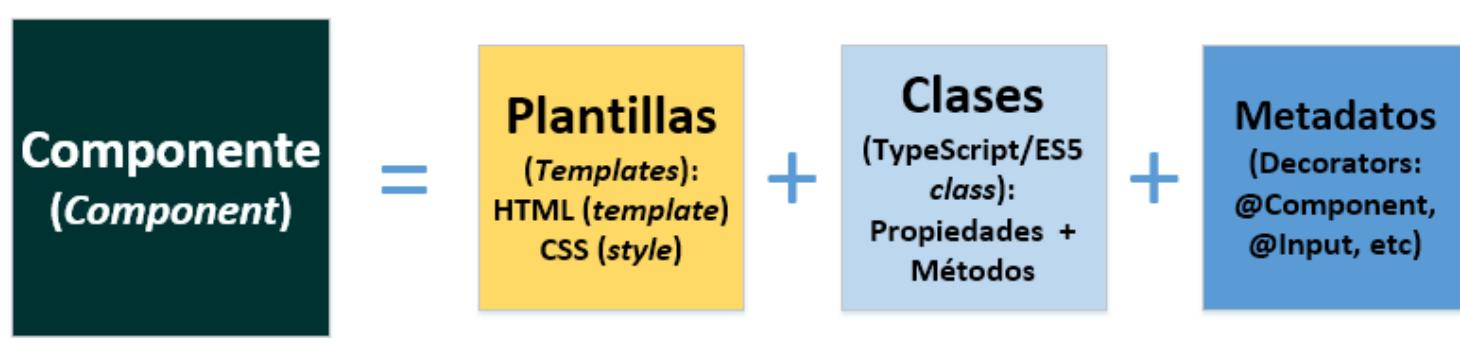
- Vamos a plantear una aplicación simple, de gestión de productos, y vamos a ir viendo a través de su construcción los distintos aspectos de Angular 2

## Arquitectura de la aplicación de Productos



# Estructura de un componente

- Recordemos el esquema inicial de componentes:



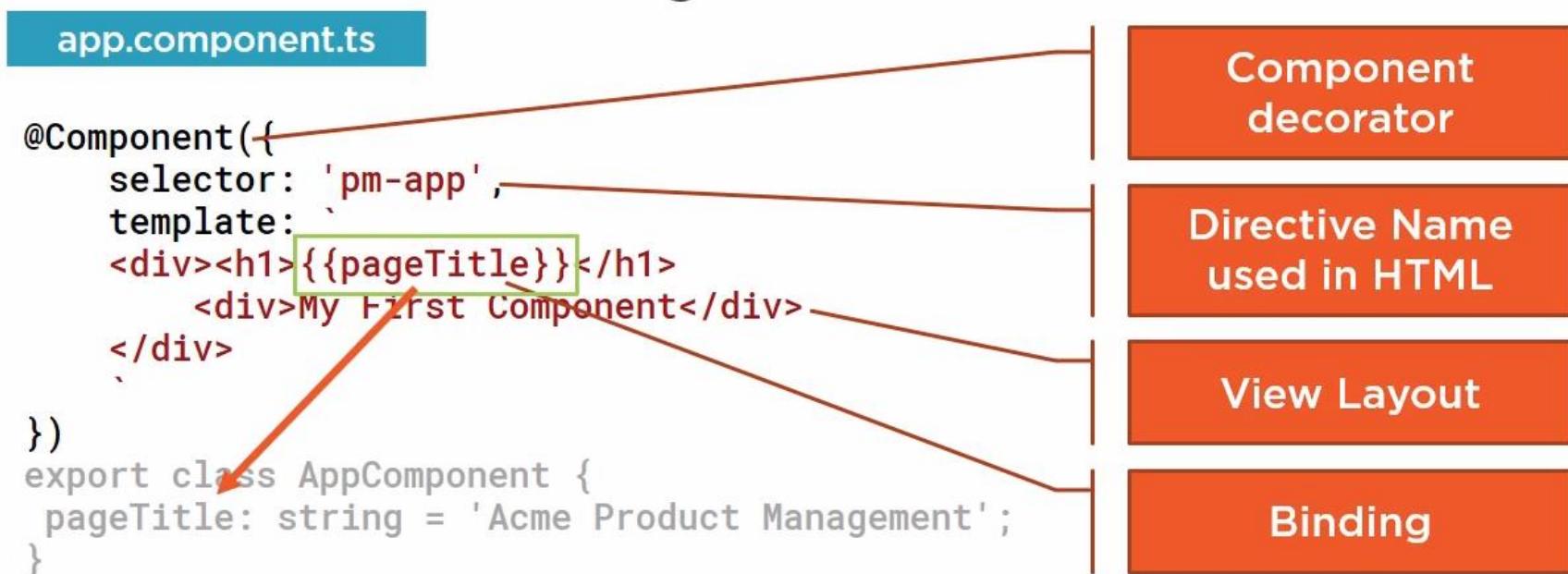
- La plantilla pueden contener "bindings" para enlazar con datos almacenados en la clase (usando sintaxis mustache {{dato}}) así como otras directivas definidas por Angular.
- Una clase, que aporta los datos que debe enlazar la IU y la parte funcional (dispone de propiedades y métodos).
- Metadatos (decorators), que configuran el comportamiento de ambos y su relación entre sí, y los registra en el "motor" de Angular.

# Decorators

- Un decorador no es más que una función que añade metadatos a una clase, un método o una propiedad.
- Los decoradores, no están disponibles todavía en JavaScript (ninguna versión), aunque están propuestas para la siguiente (ES7).
- Siempre se programan con el prefijo @
- Algunos, están incluidos en Angular de manera predeterminada:
  - @Component
  - @NgModule
  - @Router
  - @Injectable
  - @Inject
  - @View
  - @Input
  - Etc...

# Decorators

- › Los elementos que componen un decorator podemos analizarlos en el siguiente esquema de código:



- › Finalmente, para que `@Component` sea reconocido, utilizamos en la aplicación inicial la sentencia:

- › `import { Component } from '@angular/core'`

# El componente inicial de la primera aplicación

- Podemos basarnos en la aplicación quickstart, copiar su contenido en otro directorio, y cambiar algunas cosas para adaptarlas a nuestro proyecto.
- Además, así repasamos la cadena de control, y probamos alguna funcionalidad extra.
- La demo app\_productos\_1 es el punto de partida con unos elementos básicos cambiados y adaptados a la nueva propuesta.
- Hay una carpeta "recursos" que sirve para almacenar los datos externos que maneje la aplicación.
  - Iconos, archivos de datos .json, gráficos, fuentes, etc.
- Hemos añadido el favicon de la página (aunque en Edge podría no verse desde localhost sino solo en producción (dependiendo de la versión)).
- En IE10+ y en resto de navegadores, se ve perfectamente también desde el modo desarrollo.

# El componente inicial de la primera aplicación

- Podemos basarnos en la aplicación quickstart, copiar su contenido en otro directorio, y cambiar algunas cosas para adaptarla a nuestro proyecto.
- También hemos añadido un elemento styles a la plantilla del componente para comprobar la sintaxis y la funcionalidad.
- Igual que sucede con la plantilla, si se incrusta directamente el código CSS se coloca entre comilla invertida (tilde francesa o "back-tick": ` ).
- Eso permite escribir mas de una línea de definición. (también podríamos haber incluido una etiqueta <style> dentro de la propia plantilla)

# El componente inicial de la primera aplicación

- Estos estilos solo se aplican al componente donde se definen.
- Esto evita colisiones de nombres con otras definiciones (propias o ajenas)
- Además, los cambios de estilo en otras zonas de la página no afectan a lo definido por estos estilos.
- Por supuesto, la forma alternativa consiste en indicar un archivo separado para la parte HTML mediante templateUrl y lo mismo para los archivos CSS, mediante la definición styleUrls.
  - Estos enlaces serían relativos a la raíz del sitio, no al path del componente que los define.
  - También podemos usar la directiva @import 'fichero.css';



## Pongámoslo en práctica

- Modifica TareasApp para que tenga un componente con 2 partes:
  - Lista de tareas: id, descripción, tiempo, id\_proyecto
  - Lista de proyectos: id, nombre
  
- De momento haz que las dos listas se muestre en una página

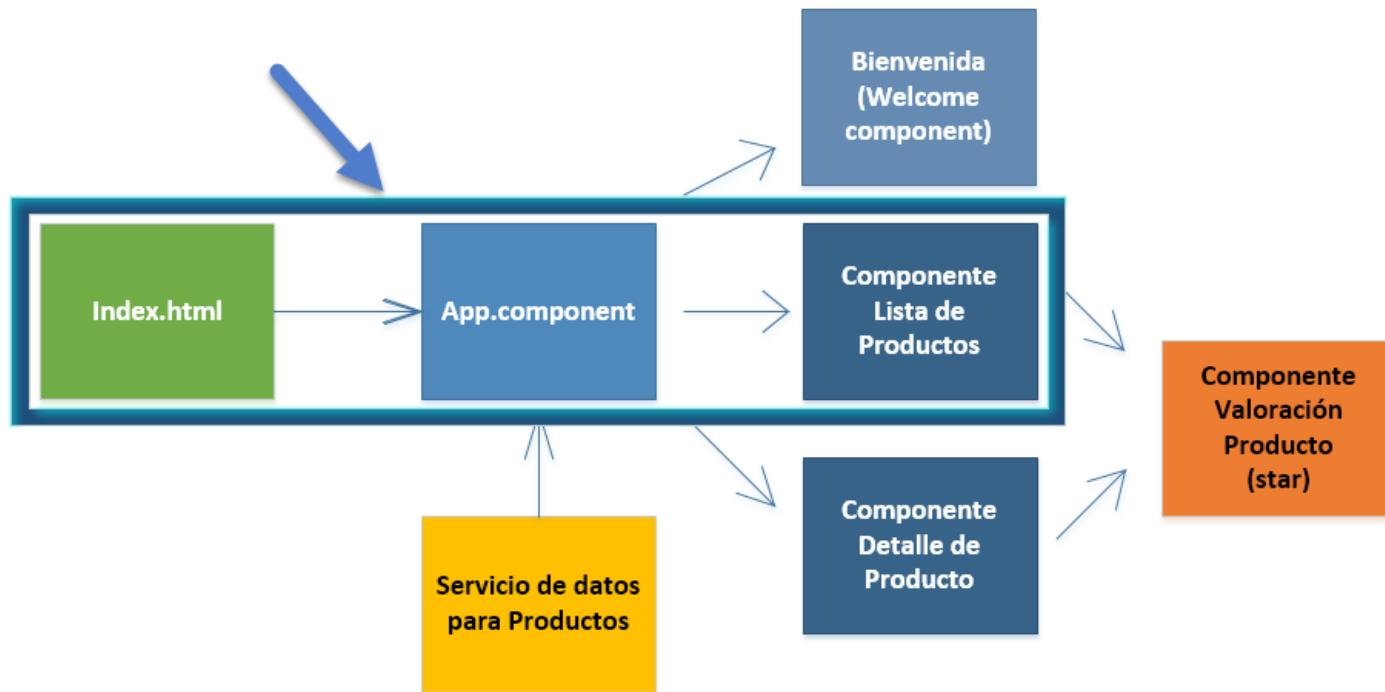
# 4

## TEMPLATES, INTERPOLACIÓN Y DIRECTIVAS

# Objetivos del módulo

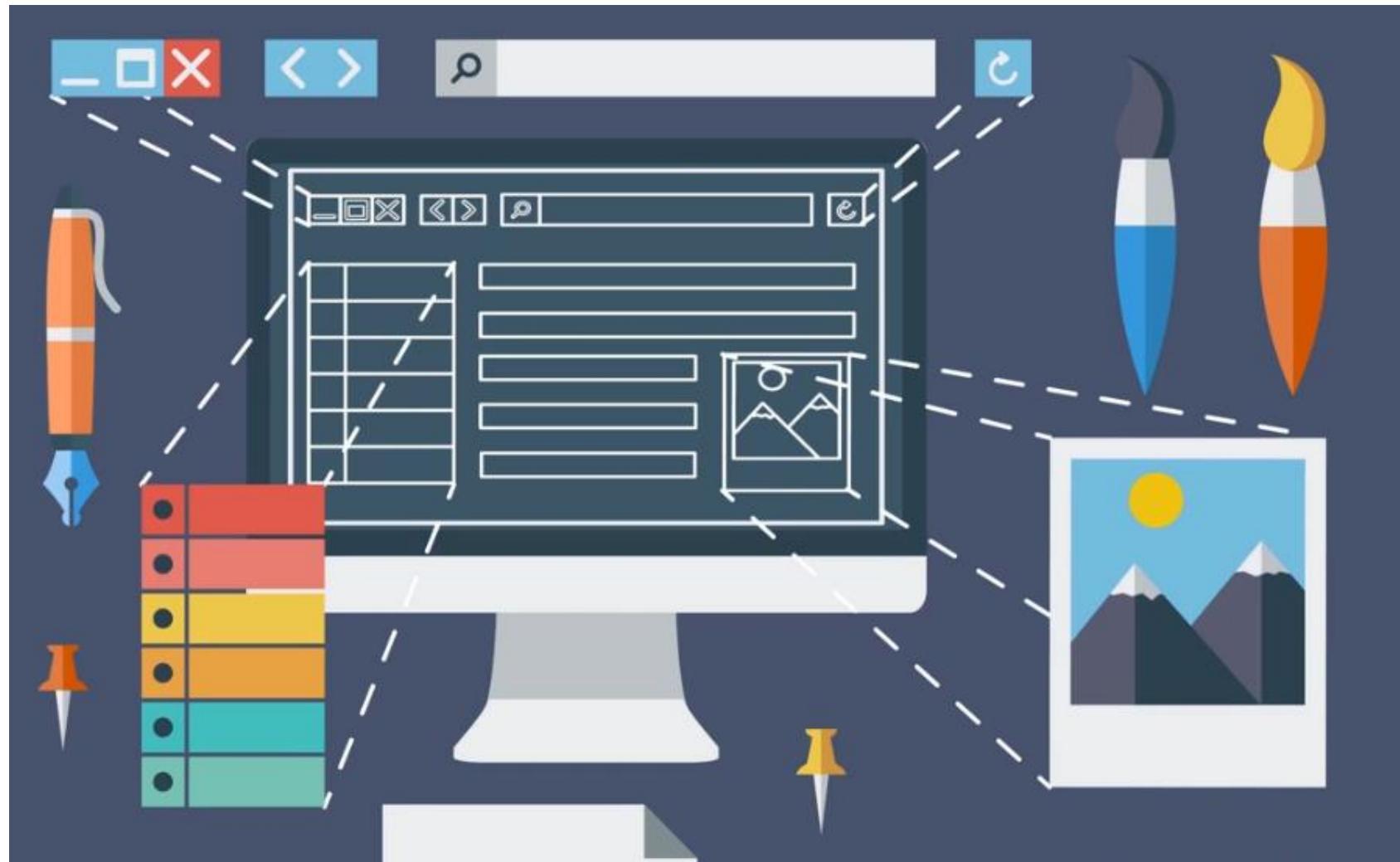
- Continuamos con la construcción de nuestra aplicación. Ahora tenemos que añadir algún contenido. En concreto, una lista de productos.
- Al hacerlo, veremos algunos aspectos de las plantillas, y las directivas

## Arquitectura de la aplicación de Productos



# Estructura de un componente

- Al final la página es un conjunto de elementos de IU



# Construcción de plantillas

- Hemos visto, que podemos distinguir 3 formas de creación de plantillas:

## Inline Template

```
template:  
"<h1>{{pageTitle}}</h1>"
```

## Inline Template

```
template:  
'  
<div>  
  <h1>{{pageTitle}}</h1>  
  <div>  
    My First Component  
  </div>  
</div>  
'
```

## Linked Template

```
templateUrl:  
'product-list.component.html'
```

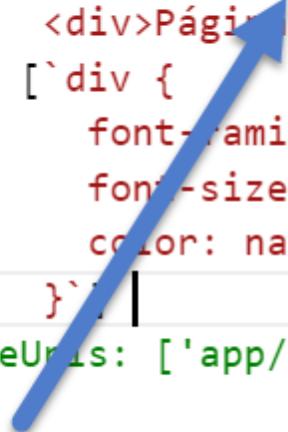
ES 2015  
Back Ticks

- Aunque las dos primeras son útiles, en proyectos reales tenderemos a utilizar la 3<sup>a</sup> forma.

# Construcción de plantillas

- El primer cambio, tiene que ver con la idea de interpolación que ya estaba disponible en Angular 1.x mediante la sintaxis moustache.
- Si añadimos una propiedad a la clase definida, podemos interpolar su contenido desde la plantilla utilizando moustache:

```
@Component({  
    selector: 'gestion-productos',  
    template: `<h1>{{tituloPagina}}</h1>  
              <div>Página de bienvenida (modificada)</div>`  
,styles: [`div {  
        font-family: "Segoe UI", Arial;  
        font-size:1.5em;  
        color: navy;  
    }`]  
    // ,styleUrls: ['app/estilografico.css']  
})  
export class AppComponent {  
    tituloPagina: string = "Demo de Gestión de Productos";  
}
```



# Cambios en la estructura

- Vamos a utilizar BootStrap para el formato de las páginas, por lo que añadimos la entrada correspondiente en **index.html**:

```
<link href="node modules/bootstrap/dist/css/bootstrap.css" rel="stylesheet" />
```

- Podemos además, añadir alguna clase predeterminada de Bootstrap para garantizar que se añaden márgenes, se selecciona un tipo de letra más legible y todo queda preparado para una configuración posterior.

```
<body class="container">
```

- Deberíamos apreciar algunos cambios, además de comprobar que se integra perfectamente con el resto (y es *responsive*).

Demo de Gestión de  
Productos

Página de bienvenida (modificada)

# La vista HTML

- Por convención se utiliza una distribución de directorios que sigue los principios DDD (Domain Driven Design):
- Cada apartado principal de la aplicación dispone de su propio directorio, dentro del directorio **app**. Definimos a este dominio como **lista-productos**.
  - Por tanto, creamos un directorio "**app/productos**", donde alojaremos todas las vistas y otros elementos relacionados con ellos.
  - Dentro de ese directorio, creamos una plantilla para el componente, que –por convención- tiene el mismo nombre que el componente con la extensión HTML.
  - Llamamos a esta vista **lista-productos.component.html**
    - Puedes ver el código fuente en la demo **app.productos\_2**

# El componente

- Según lo indicado, lo siguiente es crear el componente, que –por la misma convención- llamamos **lista-productos.component.ts**
- La técnica a seguir es la misma, solo que hay algunas cosas a considerar para **integrar** el componente dentro del resto.

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

@Component({
    selector: 'lista-productos',
    templateUrl: 'app/productos/lista-productos.component.html'
})
export class ListaProductosComponent {

}
```

- Igual que antes, definimos un selector (**lista-productos**) y una plantilla, que esta vez hemos escrito en un fichero separado.

# Enlazando componentes

- Para que nuestro componente y su plantilla sean reconocidos por el componente que los aloja, declaramos su selector en el propio componente principal.
- Así que, el decorador del componente principal pasa a ser:

```
@Component({  
  selector: 'gestion-productos',  
  template: `|  
    <h1>{{tituloPagina}}</h1>  
    <lista-productos></lista-productos>`  
})
```

- Esto enlaza el principal con el secundario, pero ahora falta indicarle al módulo que maneja **app.component** donde se encuentra nuestra lista de productos.

# Declarando el nuevo componente en su módulo

- Aquí se produjo un "breaking change" desde la RC4 hacia arriba (hasta la versión final).
- Hasta ese momento, el componente era declarado con una entrada **directives** dentro de su decorador contenedor.
- A partir de ese momento, esas declaraciones se separan al **modulo** que maneja el componente principal, de manera que el módulo principal queda de la siguiente forma:

```
import { NgModule }      from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { AppComponent }  from './app.component';
import { ListaProductosComponent } from './productos/lista-productos.component';

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



# El aspecto final hasta este punto

- El aspecto final si visualizamos nuestro proyecto hasta este momento en el navegador será similar a este:

Demo de Gestión de Productos

Lista de productos

Filtrar por:

Filtrado por:

Mostrar imagen      Producto      Código      Disponible      Precio      Valoración

- Donde la salida debiera adaptarse al contenedor según varía el **viewport** de presentación.
- Si quieres ver la salida adaptada a dispositivos móviles, puedes usar el emulador "**Ripple**" de Chrome o instalar el navegador **Blisk**.

# Directivas incorporadas

- Por supuesto, además de crear nuestras propias directivas, Angular incluye un conjunto de directivas incorporadas.
- Estas directivas se pueden categorizar de diversas formas:
- Estructurales (que modifican la estructura del DOM/Vista)



- Sintácticamente, se distinguen del resto por la presencia del asterisco (\*) delante de la definición de la directiva.

# Directivas incorporadas

- Podemos probar esto añadiendo esa directiva a nuestra tabla, ya que de momento, no contiene ningún elemento.
- La directiva tiene que referirse a algún objeto accesible en el componente, así que creamos una entrada **productos**, indicando que es un array de tipo **any**.

```
export class ListaProductosComponent {  
    productos: any[];  
}
```

- Y a la etiqueta **<table>**, le añadimos una directiva **\*ngIf**, para que solo muestre la tabla si existe una definición de productos y además, no está vacía.

```
<table class='table' *ngIf="productos && productos.length">
```

- Si visualizas después de esto, comprobarás que no aparece la tabla.

# Directivas incorporadas

- Podemos hacer una primera prueba de presentación de datos, que se parezca al resultado final, añadiendo elementos a esa definición de productos.
- Para ello, simplemente, añadimos un par de elementos al componente en la definición
- Solamente hay que recordar después los nombres de los campos, para incluirlos exactamente como están en la vista.

```
productos: any[] = [  
    {  
        "Producto": "Leaf Rake",  
        "Codigo": "GDN-0011",  
        "Disponible": "March 19, 2016",  
        "Precio": 19.95,  
        "Valoracion": 3.2  
    },  
    {  
        "Producto": "Garden Cart",  
        "Codigo": "GDN-0023",  
        "Disponible": "March 21, 2016",  
        "Precio": 32.99,  
        "Valoracion": 4.2,  
    }];
```

# Directivas incorporadas

- También podemos usar la directiva **\*ngFor** de manera similar a como lo hacíamos en versiones anteriores, con la directiva **ng-repeat**
  - Por cierto, observa el cambio de nomenclatura de la separación por guiones "**kebab-case**", a la actual, "**camelCase**"
- Por ejemplo, añadimos una entrada global de fila a la tabla y le indicamos que vaya iterando por la colección y creando tantos elementos como tenga el array de datos.

```
<tbody>
  <tr *ngFor="let producto of productos">
    <td></td>
    <td>{{producto.Producto}}</td>
    <td>{{producto.Codigo}}</td>
    <td>{{producto.Disponible}}</td>
    <td>{{producto.Precio}}</td>
    <td>{{producto.Valoracion}}</td>
  </tr>
</tbody>
```

# Salida final del primer listado

- Con todos esos cambios, la salida final del listado tendría un par de elementos solamente, pero ya podemos hacernos una idea del resultado final.
- Debería tener un aspecto similar al siguiente:

Demo de Gestión de Productos

Listado de productos

Filtrar por:

Filtrado por:

Mostrar imagen	Producto	Código	Disponible	Precio	Valoración
	Leaf Rake	GDN-0011	March 19, 2016	19.95	3.2
	Garden Cart	GDN-0023	March 21, 2016	32.99	4.2



## Pongámoslo en práctica

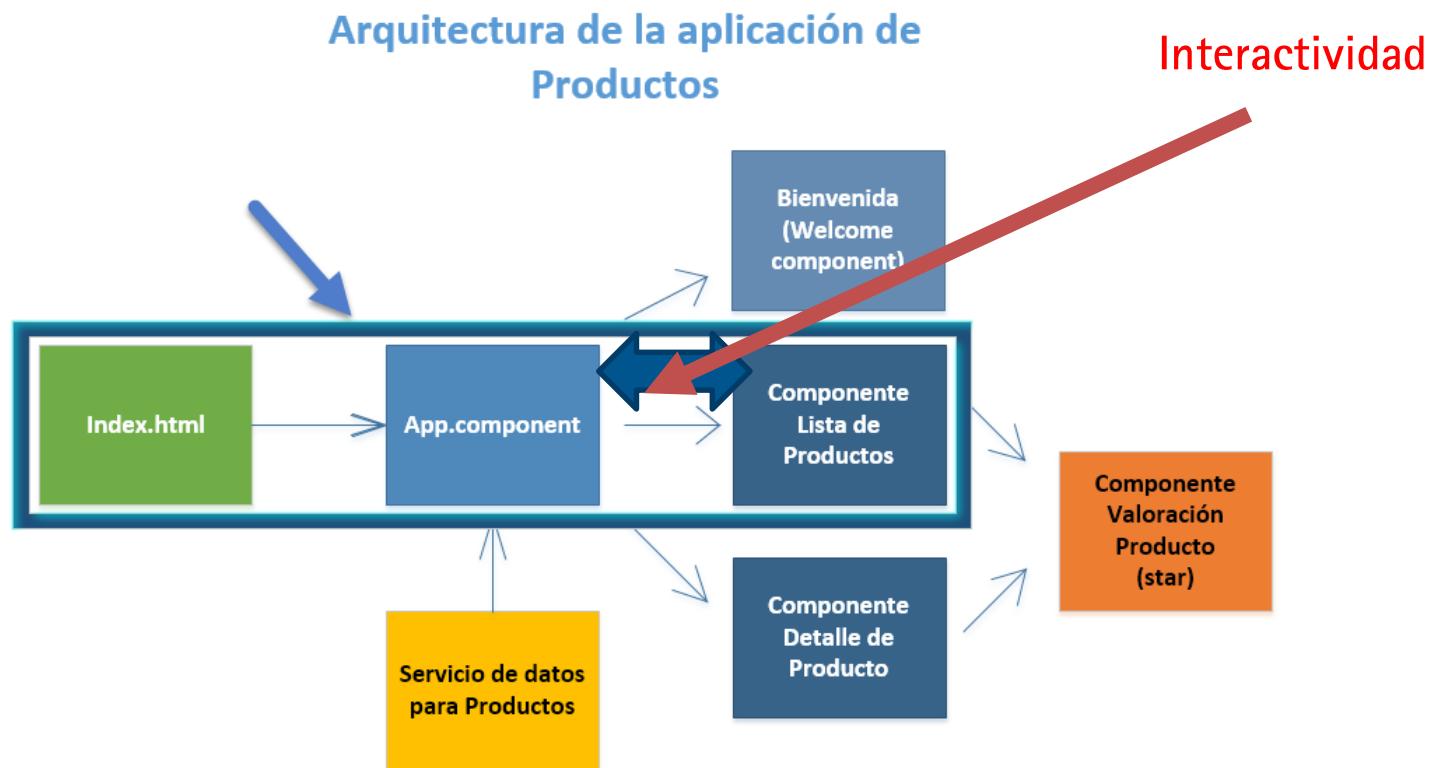
- Separa los componentes en dos archivos que se incluyan desde el componente principal.
- Haz que las dos listas se muestre en una página

5

# DATA BINDING Y PIPES

# Interactividad entre el DOM y los componentes

- Ya tenemos una lista de productos. Pero no existe ningún tipo de interactividad entre el DOM y los componentes
- Vamos a añadir esos factores para incorporar eventos y otros aspectos.



# Interactividad entre el DOM y los componentes

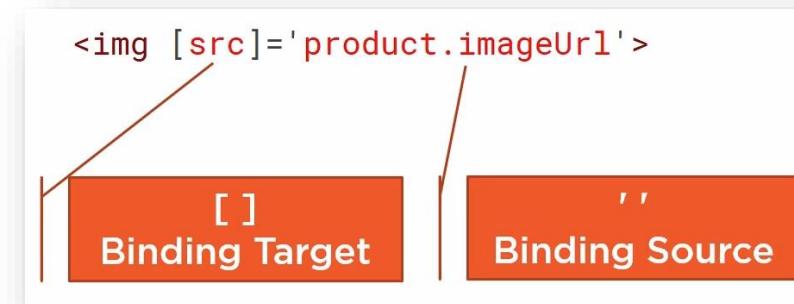
- Ya tenemos una lista de productos. Pero no existe ningún tipo de interactividad entre el DOM y los componentes
- Vamos a modificar los datos, de forma que incluyan una imagen y podamos ver el funcionamiento de los gráficos y dar cobertura al botón correspondiente.
- La manera en que el *databinding* se ha modificado en esta versión ha sufrido dos modificaciones importantes respecto a versiones 1.x
- Por un lado se mantiene la posibilidad de usar la sintaxis *moustache* como antes (mecanismo de interpolación), y por otro, se puede expresar un *databinding* poniendo entre paréntesis la propiedad a enlazar y entre comillas el dato de destino.
- Por ejemplo para enlazar con un gráfico podemos seguir utilizando interpolación:

```

```

# Databinding

- Pero también disponemos de una alternativa propia de Angular 2, en la que la propiedad a enlazar tiene este aspecto:



- De modo que la sintaxis anterior quedaría de esta forma:

```
<img [src]="producto.imageUrl" style="width:50px"/>
```

- De modo que si modificamos los datos del componente para que incluya un par de referencias a imágenes, tendremos que tener la salida modificada incluyendo los gráficos.
- Aunque se prefiere el **databinding**, si tenemos que usar una expresión compuesta, la interpolación puede ser la solución más adecuada:

```
<img src='http://openclipart.org/{{product.imageUrl}}'>
```

# Databinding

- También podemos establecer un mecanismo de **binding** sobre propiedades de una clase.
- En este caso, la anchura y el margen de la propiedad gráfica de cada producto, se prestan bien para ser expresadas de esa forma:

```
export class ListaProductosComponent {  
    imageWidth: number = 50;  
    imageMargin: number = 3;  
    productos: any[] = [  
        {  
            "Producto": "Leaf Rake",  
            "Codigo": "GDN-0011",  
            "Disponible": "March 19, 2016",  
            "Precio": 19.95,  
            "Valoracion": 3.,  
            "imageUrl": "http://openclipart.org/  
    },  
    ]
```

# Databinding

- Y el correspondiente código fuente de la plantilla, quedaría así:

```
<img [src]="producto.imageUrl"
      [title]="producto.Producto"
      [style.width.px]="imageWidth"
      [style.margin.px]="imageMargin" >
```

- Observa que para los estilos, no ponemos el nombre del objeto, por que son propiedades de la clase con al que enlazamos.
- El mecanismo de **Databinding** (igual que la interpolación) va en un solo sentido (**one-way**).
- Ahora nuestra demo tiene que aparecer correctamente, aunque el botón de ocultar las imágenes, (obviamente) no funciona, porque tenemos que incluir el manejador de eventos correspondiente.

# Databinding

- En la demo se incluyen las dos versiones, para comprobar que son equivalentes en este caso.

## Demo de Gestión de Productos

Lista de productos

Filtrar por:

Filtrado por:

Mostrar imagen



Garden Cart



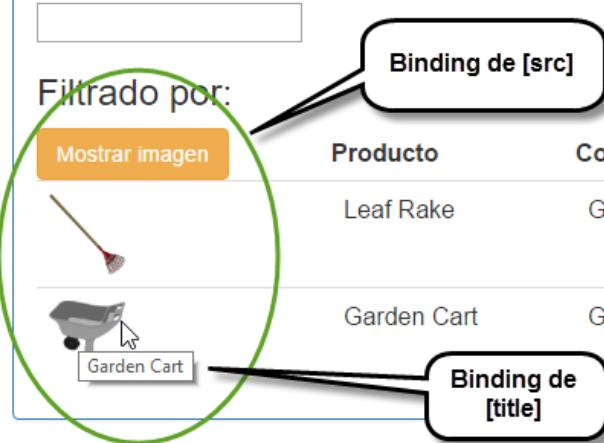
Leaf Rake

Garden Cart

Producto	Codigo	Disponible	Precio	Valoracion
Leaf Rake	GDN-0011	March 19, 2016	19.95	3
Garden Cart	GDN-0023	March 21, 2016	32.99	4.2

Binding de [src]

Binding de [title]



# Vinculación con eventos

- En la parte de los eventos la sintaxis es similar, pero el evento correspondiente se encierra entre paréntesis. De modo que modificaremos el código del botón de la siguiente forma:

```
<button class="btn btn-warning"
        (click)="mostrarImagen()>
    Mostrar imagen
</button>
```

- A continuación, crearemos una propiedad y un método en la clase del componente:

```
imageVisible: boolean = false;

mostrarImagen(): void {
    this.imageVisible = !this.imageVisible;
}
```

- Por último estableceremos que la imagen es visible solo condicionalmente:

```
<img *ngIf="imageVisible"
      [src]="producto.imageUrl"
```

# Expresiones de interpolación

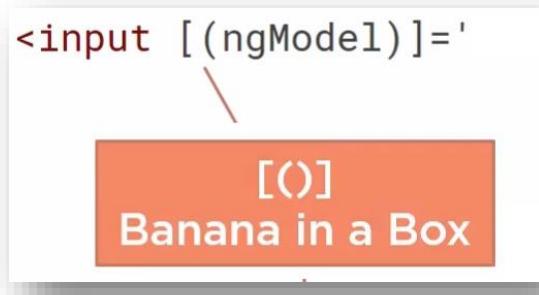
- Nos queda hacer que el texto del botón también se modifique cuando pulsemos sobre él, cambiando el texto de "Mostrar Imagen" a "Ocultar Imagen".
- Esto es sencillo: cambiamos el texto del botón por una interpolación:

```
<button class="btn btn-warning"
        (click)="mostrarImagen()">
    {{imageVisible ? "Mostrar" : "Ocultar"}} imágenes
</button>
```

- Recordemos que la interpolación no solo lee datos, sino que evalúa expresiones.

# Binding bidireccional (two-way binding)

- Si queremos que la entrada de la caja de texto muestre el criterio de ordenación, tenemos que hacer que, al introducir una expresión de filtro, ésta aparezca en la etiqueta inferior.
- Esto se realizaba con la directiva **ng-model** en la versión anterior, y en ésta es muy similar pero se usa la sintaxis "*banana in a box*".
- El nombre es debido a la sintaxis que se utiliza:



- Los corchetes externos indican enlace a propiedad (la que se indique después del signo igual), mientras los paréntesis internos indican evento (cualquier entrada que cambie los datos a los que apunta)

# Binding bidireccional (two-way binding)

- Ahora bien, si queremos este tipo de funcionalidad en elementos de entrada, tenemos que modificar nuestro módulo, para importar esta operativa desde el bloque "*FormsModule*"
- Este bloque contiene esa y otras funcionalidades propias de los mecanismos de entrada y su paso al modelo de datos.
- Por tanto, deberemos volver a la defunción de nuestro módulo, y modificarlo de la siguiente forma:

```
import { NgModule }      from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { FormsModule } from '@angular/forms'; [
import { AppComponent } from './app.component';
import { ListaProductosComponent } from './productos/lista

@NgModule({
  imports:      [ BrowserModule, FormsModule ],
  declarations: [ AppComponent, ListaProductosComponent ],
  bootstrap:   [ AppComponent ]
})
```

# Binding bidireccional (two-way binding)

- Ahora, podremos cambiar nuestro mecanismo de filtro, que contendrá en la plantilla esta sintaxis (**ngModel** se define en **FormsModule**):

```
<div class="row">
    <div class="col-md-2">Filtrar por:</div>
    <div class="col-md-4">
        <input type="text"
            [(ngModel)]='filtro' >
    </div>
</div>
<div class="row">
    <div class="col-md-6">
        <h3>Filtrado por: {{filtro}}</h3>
    </div>
</div>
```

- Y definimos la propiedad '**filtro**' en nuestra clase:

```
filtro: string = "cart";
```

# Binding bidireccional (two-way binding)

- Con lo que –nuevamente– tendremos una pantalla inicial que muestra un filtro predeterminado en la caja de texto, y el "eco" de esa propiedad en el elemento **<h3>** ("Filtrado por:") de la plantilla.
- Si cambias el valor del **input** de entrada, verás como se hace eco automáticamente en la propiedad:

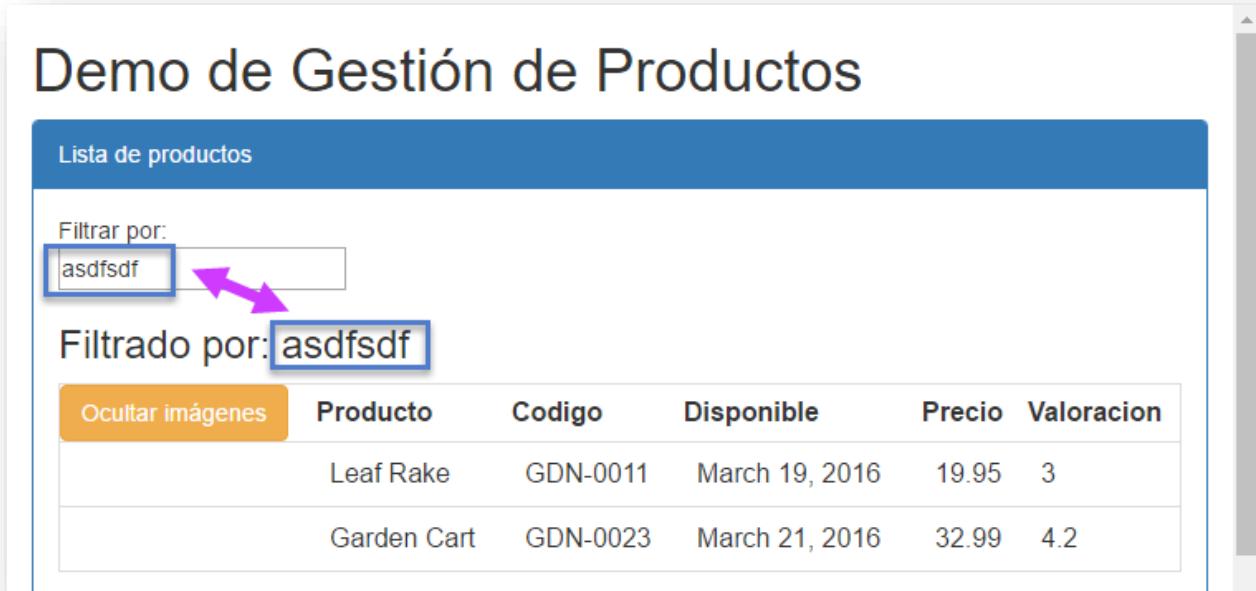
Demo de Gestión de Productos

Lista de productos

Filtrar por: asdfsdf

Filtrado por: asdfsdf

Ocultar imágenes	Producto	Codigo	Disponible	Precio	Valoracion
	Leaf Rake	GDN-0011	March 19, 2016	19.95	3
	Garden Cart	GDN-0023	March 21, 2016	32.99	4.2



# Pipes (Filtros de salida)

- Otro aspecto similar al que existía en Angular 2 son los filtros de salida, antes llamados *filters*, y ahora **pipes** por el símbolo utilizado en su sintaxis.
- Se escriben a continuación de una expresión y modifican la manera es que ésta se interpreta en el DOM, sin modificar en ningún momento el dato de entrada.

```
 {{ product.productCode | lowercase }}

<img [src]='product.imageUrl'
      [title]='product.productName | uppercase'
```



# Pipes (Filtros de salida)

- Otro aspecto similar al que existía en Angular 2 son los filtros de salida, antes llamados *filters*, y ahora **pipes** por el símbolo utilizado en su sintaxis.
- Se escriben a continuación de una expresión y modifican la manera es que ésta se interpreta en el DOM, sin modificar en ningún momento el dato de entrada.

```
 {{ product.productCode | lowercase }}  
  
 <img [src] ='product.imageUrl'  
       [title] ='product.productName | uppercase'>  
  
 {{ product.price | currency | lowercase }}  
  
 {{ product.price | currency:'USD':true:'1.2-2' }}
```



# Pipes (Filtros de salida)

- Podemos probar algunos aspectos de pipes en nuestro código, cambiando la salida del código de los productos para que aparezca en minúsculas (*lowercase*), y formateando el precio para que adopte el formato de moneda (\*)

```
<td>{{producto.Producto}}</td>
<td>{{producto.Codigo | lowercase }}</td>
<td>{{producto.Disponible}}</td>
<td>{{producto.Precio | currency:'EUR':true:'1.2-2' }}</td>
<td>{{producto.Valoracion}}</td>
```

- En el caso del formato de moneda, el segundo argumento, (a diferencia de lo que sucedía en la versión anterior), es el código ISO 4217:
  - [https://en.wikipedia.org/wiki/ISO\\_4217](https://en.wikipedia.org/wiki/ISO_4217)
- El tercer argumento (*true*) indica que queremos mostrar decimales, y el cuarto significa que queremos al menos una cifra entera y al menos 2 decimales con un máximo de 2 decimales.

# Pipes (Filtros de salida)

- Al igual que sucedía con los filtros, el usuario puede crear sus propios filtros, que no son sino funciones que se registran como *pipes*.
- Otro aspecto a tener en cuenta es que la salida predeterminada de Angular 2 (y la anterior) es en inglés de EE.UU.
- Para cambiar el comportamiento predeterminado, deberemos incluir el "locale" que corresponda al país que se desee (numerónimos *i18n*).
- La salida reflejará los cambios, como es de esperar:

Demo de Gestión de Productos

Lista de productos

Filtrar por:  
cart

Filtrado por: cart

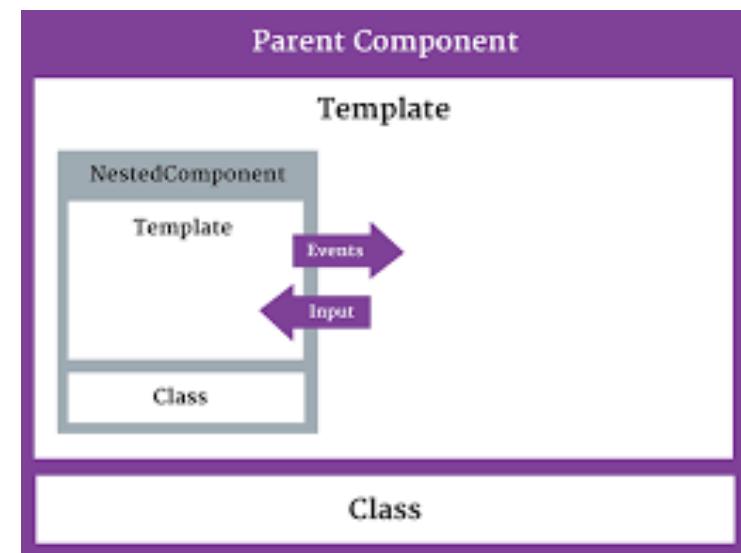
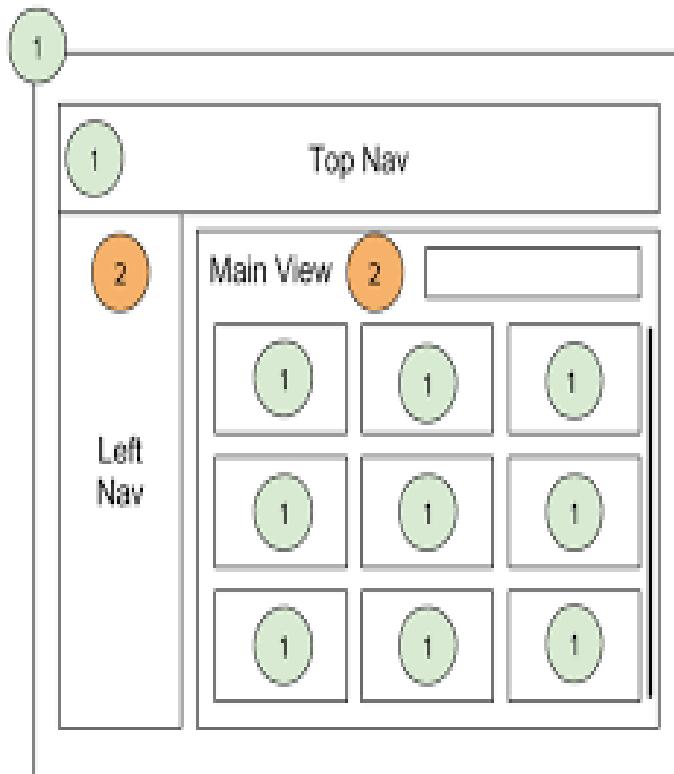
Ocultar imágenes	Producto	Código	Disponible	Precio	Valoración
	Leaf Rake	gdn-0011	March 19, 2016	€19.95	3
	Garden Cart	gdn-0023	March 21, 2016	€32.99	4.2

6

# COMPONENTES ANIDADOS

# Estructura y operativa

- En Angular 2 los componentes funcionan como las "**matrioskas**" (muñecas rusas): usan en modelo contenedor/contenido.

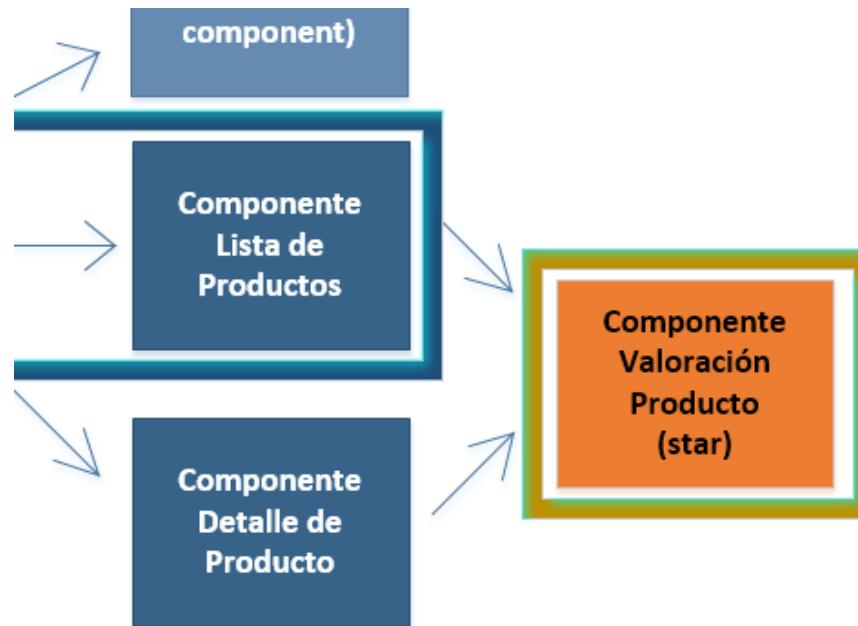


# Situaciones recomendadas

- Se recomienda declarar un componente como anidado (***nestable***) en los siguientes casos:
  - Su plantilla solo gestiona un fragmento de una vista más grande.
  - Tiene un selector, de forma que puede usarse como una directiva.
  - Opcionalmente, se comunica con su contenedor.
  - Los componentes anidados se comunican con el exterior mediante los decoradores **@input** y **@output**

# Componentes anidados

- Seguimos trabajando con componentes, esta vez analizando como se crean y manejan los componentes anidados.



- En nuestra aplicación crearemos un componente **Valoración** que presentará un gráfico de estrellas en función del valor almacenado.

# Estructura y operativa

- El componente valoración tendrá que recibir un parámetro de entrada (el **número**) y transformar ese número en las estrellas correspondientes.
- A su vez, queremos que cuando el usuario pulse sobre el componente, se emita un **evento** que notifique los posibles cambios.
- Si lo hacemos como otro componente anidado el componente podrá reutilizarse en diversas partes de la aplicación.
- Desde el punto de vista de la organización de directorios, eso supone que lo creemos en un directorio separado.
- Para seguir con las buenas prácticas, es preferible que creamos un directorio compartido ("**shared**"), y situemos ahí el componente, así que, el primer paso, será la creación del directorio (al mismo nivel que "**productos**").

# Plantillas del componente

- La parte de la interfaz de usuario se compone de dos archivos: la plantilla HTML (***shared.component.html***) y la hoja de estilo (***shared.component.css***).
- Situamos ambas en la carpeta "***shared***".
- La hoja de estilo es muy sencilla, porque la mayor parte de la funcionalidad visual la va a aportar Bootstrap.
- Así que el contenido se reduce a lo siguiente:

```
.crop {  
    overflow: hidden;  
}  
div {  
    cursor: pointer;  
}
```

- La clase ***crop*** se encargará de que no se muestre contenido que exceda el espacio disponible.

# Plantillas del componente

- La plantilla HTML (**shared.component.html**) hace referencia a Bootstrap y lo visto antes en la clase.

```
<div class="crop"
    [style.width.px]="starWidth"
    [title]='rating'
    (click)='onClick()'>
    <div style="width: 86px">
        <span class="glyphicon glyphicon-star"></span>
        <span class="glyphicon glyphicon-star"></span>
        <span class="glyphicon glyphicon-star"></span>
        <span class="glyphicon glyphicon-star"></span>
        <span class="glyphicon glyphicon-star"></span>
    </div>
</div>
```

- Observa cómo se utilizan los iconos manejados por **BootStrap**, y cómo se definen dos *binding* de atributos y uno del evento **click**.

# Plantillas del componente

- La plantilla define dos propiedades para *binding*: ***starWidth***, que determina la anchura (incluso fraccionaria), y ***rating***, que se corresponde con el valor numérico a mostrar.
- También define un *event-binding* para **click** (como respuesta a interacciones IU)

The diagram shows the template code for a component with annotations:

```
<div class="crop"
    [style.width.px]='starWidth'
    [title]='rating'
    (click)='onClick()'
    <div style="width: 86px">
        <span class="glyphicon glyphicon-star"></span>
        <span class="glyphicon glyphicon-star"></span>
        <span class="glyphicon glyphicon-star"></span>
        <span class="glyphicon glyphicon-star"></span>
        <span class="glyphicon glyphicon-star"></span>
    </div>
</div>
```

- Annotations highlight the following:
  - [style.width.px] = 'starWidth' (highlighted in blue)
  - [title] = 'rating' (highlighted in blue)
  - (click) = 'onClick()' (highlighted in red)
- A callout bubble labeled "2 Propiedades" points to the first two highlighted lines.
- A callout bubble labeled "1 evento" points to the third highlighted line.

- Estas propiedades tendrán que reflejarse en la estructura de la clase, así como el evento definido.

# Construcción del componente anidado

- Al componente, lo llamamos (por seguir una nomenclatura conocida) "**star**", de forma que el archivo será: "**star.component.ts**".
- El contenido será similar al de los anteriores, pero con las referencias a los archivos de la IU, más sus definiciones.
- El componente, además, tiene que hacer referencia al mecanismo de evento **OnChanges**, para responder a los cambios del componente.
- Haremos que la anchura visual sea proporcional al número mediante un cálculo sencillo dentro de la clase.
- En resumen, en una primera fase:
  - Tenemos que importar **OnChanges** (para detección de cambios en valores de entrada)
  - Declarar el selector, la plantilla y los estilos
  - Declarar la clase y sus propiedades (para soporte del **binding**). Inicialmente, podemos asignar valores para contar con datos de inicio.
  - Implementar el método **ngOnChanges** que se haga cargo de la conversión valor.numérico => ancho.visual.

# Código del componente

➤ De forma que código resultante tendrá este aspecto:

```
import { Component, OnChanges, Input,
         Output, EventEmitter } from "@angular/core";

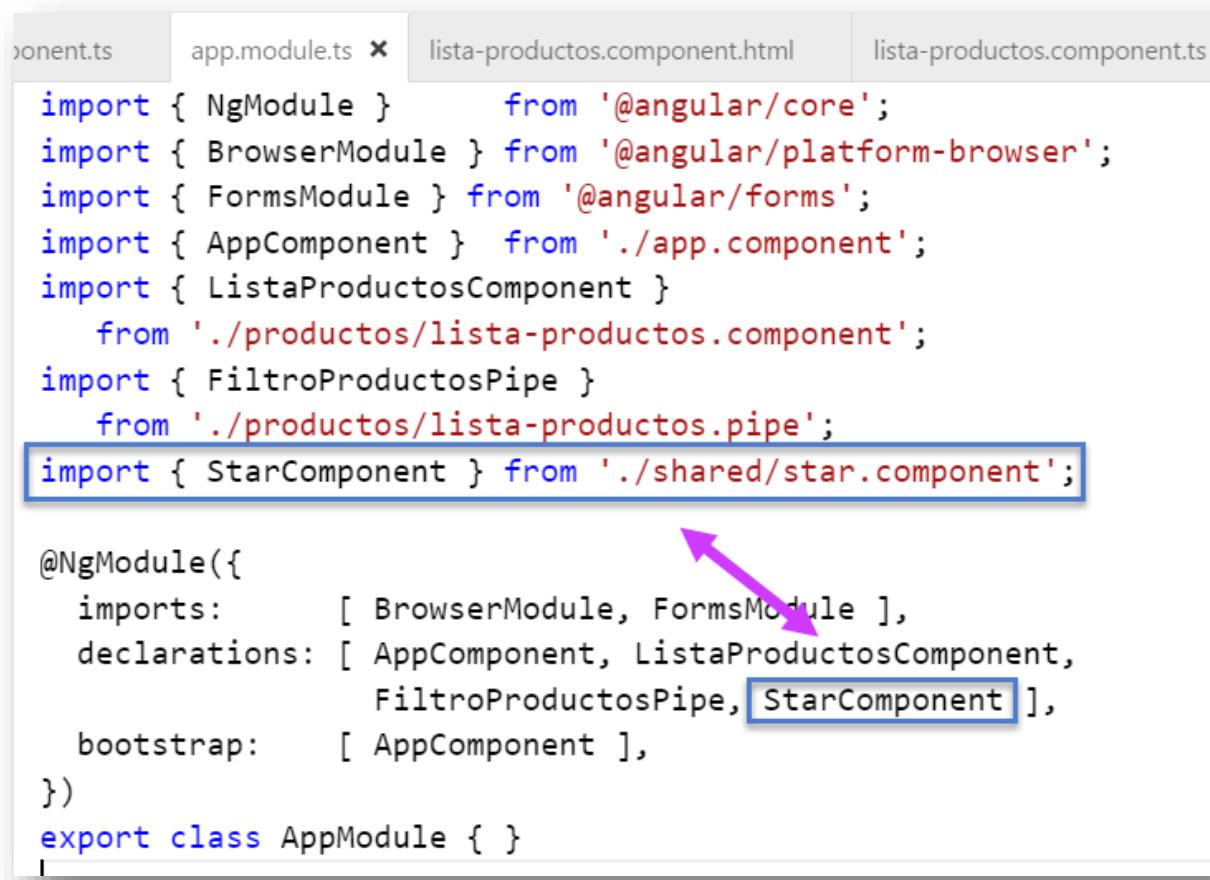
@Component({
  selector: 'ai-star',
  templateUrl: 'app/shared/star.component.html',
  styleUrls: ['app/shared/star.component.css']
})
export class StarComponent implements OnChanges {
  rating: number = 3;
  starWidth: number;
  ngOnChanges(): void {
    this.starWidth = this.rating * 86 / 5;
  }
}
```

# Código del componente

- Con esto, tenemos una primera versión del componente, aunque no posee ningún tipo de interacción con el exterior.
- Para que el componente sea utilizado por otros, debe ser declarado ***en el módulo*** que les maneja a todos (app.module.ts).
  - A partir de la RC5, se produjeron cambios importantes aquí también. En RC's anteriores, esa declaración se hacía sobre el componente que actuaba como contenedor, mediante una declaración ***directives***.
  - ***Esa declaración ha desaparecido en la versión final***
  - Ha sido sustituida por una declaración más que tendremos que añadir al módulo manejador.

# Código del componente

➤ Así que el módulo en este punto tendrá este contenido:



```
component.ts      app.module.ts ✘    lista-productos.component.html    lista-productos.component.ts
import { NgModule }      from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { FormsModule } from '@angular/forms';
import { AppComponent }  from './app.component';
import { ListaProductosComponent }
  from './productos/lista-productos.component';
import { FiltroProductosPipe }
  from './productos/lista-productos.pipe';
import { StarComponent } from './shared/star.component';

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

# Código del componente

- A su vez, la plantilla exterior, deberá utilizar el selector del componente anidado, en vez de la referencia "antigua":

```
<td>{{producto.Producto}}</td>
<td>{{producto.Codigo | lowercase }}</td>
<td>{{producto.Disponible}}</td>
<td>{{producto.Precio | currency:'EUR':true:'1.2-2' }}</td>
<td><ai-star [rating]='producto.Valoracion'
```

- Con </ai-star></td> os ver la valoración convertida en asteriscos, pero el componente no recibe datos ni los emite al exterior.
  - Para que el componente hable con el exterior, necesitamos marcar las propiedades (o eventos) del componente con **Decoradores de Comunicación**.

# Decoradores de comunicación (@Input, @Output)

- Los Decoradores de Comunicación, nos permiten marcar propiedades con metadatos que permiten exponer su contenido al exterior y/o definir que propiedades se consideran valores de entrada para el componente.
- Para ello decoramos los elementos cuyo valor es recibido del exterior mediante el decorador **@Input**

```
export class StarComponent {  
  @Input rating: number;  
  starWidth: number;  
}
```

- Una vez hecho esto, podemos pasar datos a ese elemento mediante "*property binding*" en la plantilla del contenedor.
- De esa manera el valor de la propiedad "**Valoracion**" es recibido por el componente anidado en su propiedad "**rating**".

```
<ai-star [rating]='producto.Valoracion'
```

# Decoradores de comunicación (@Input, @Output)

- Si queremos que el componente anidado comunique información al exterior, debemos usar el decorador "**@Output**".
- La manera de declarar esta situación en el componente, es mediante la definición de un evento, representado por la clase **EventEmitter**.
- **EventEmitter** permite el uso de genéricos, de manera que podemos definirlo mediante esa sintaxis e indicar el tipo de datos que queremos pasar al contenedor.
- En este caso, nos basta con comunicarlo al exterior mediante una cadena que informe a la IU qué elemento ha sido seleccionado (**clicked**), para que responda como proceda.
- En el código añadiremos la definición del evento de esta forma:

```
@Output() ratingClicked: EventEmitter<string> =  
    new EventEmitter<string>();
```

- (Observa la similitud con la sintaxis de C#)

# Decoradores de comunicación (@Input, @Output)

- Además, deberemos importar las definiciones de Output e EventEmitter al componente anidado.
- Finalmente, añadiremos el evento **onClick** para indicar el comportamiento que queremos en tiempo de ejecución.
- En resumen:

```
import { Component, OnChanges, Input,
         Output, EventEmitter } from "@angular/core";
```

```
@Input() rating: number;
starWidth: number;
@Output() ratingClicked: EventEmitter<string> =
    new EventEmitter<string>();
```

```
onClick() {
    this.ratingClicked.emit(`El usuario ha valorado ${this.rating}`);
}
```

# Decoradores de comunicación (@Input, @Output)

- En la interfaz de usuario exterior (el listado), haremos referencia a este evento (interior), mediante:

```
<td><ai-star [rating]='producto.Valoracion'  
          (ratingClicked)='onRatingClicked($event)'>  
</ai-star></td>
```

- Y en su contenido, ***onRatingClicked*** estará definido igualmente:

```
onRatingClicked(message: string): void {  
    alert(message);  
}
```

# Decoradores de comunicación (@Input, @Output)

- Con ello, la salida será similar a esto:



- De momento los datos siguen "empotrados" en la clase. El paso siguiente será analizar cómo creamos un servicio que suministre la información.



## Pongámoslo en práctica

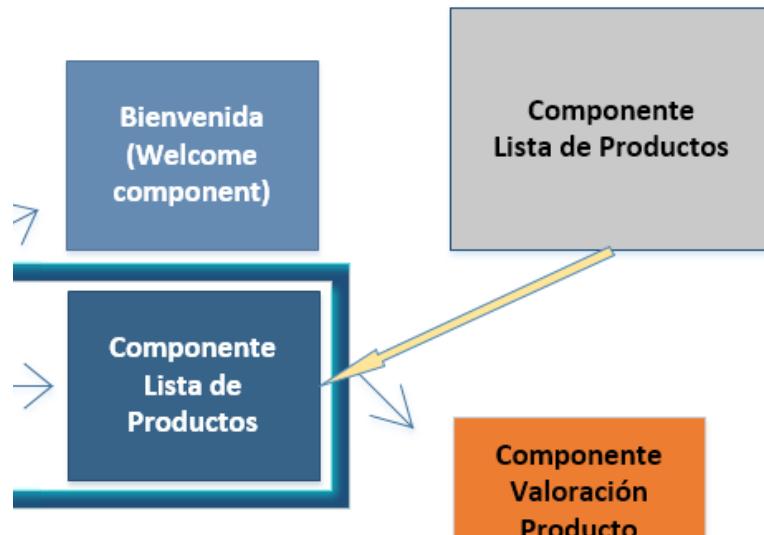
- Añade un componente anidado a cada fila de la lista de tareas. Este componente debe mostrar un botón que al ser clicado elimine la tarea correspondiente.

7

# EXTENSIÓN DE COMPONENTES

# Extensión de componentes

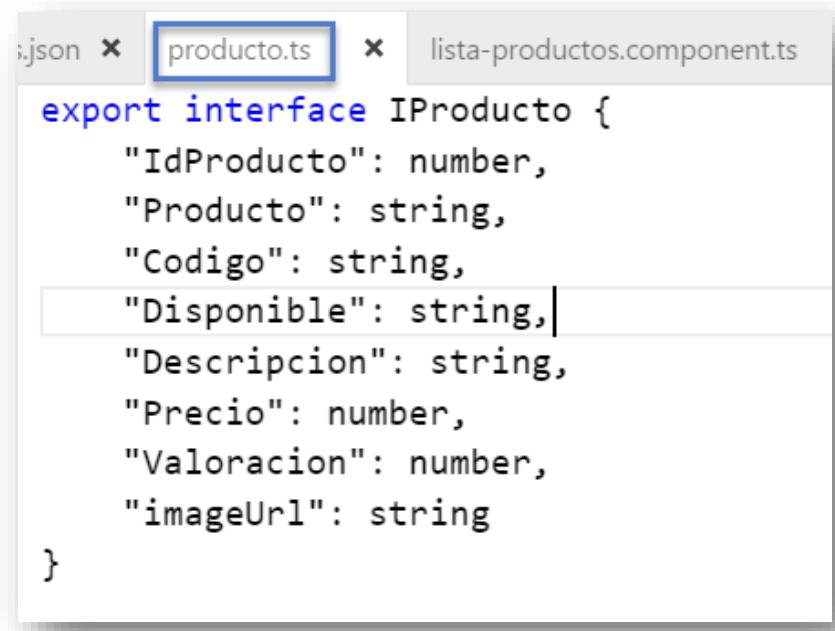
- Vamos a continuar analizando los componentes, y veremos cómo extender su funcionalidad básica para realizar otras tareas.



- Para ello, nos beneficiaremos de ciertas características de TypeScript, como **strong typing**.
- Las interfaces permiten definir contratos que, aquellos que las implementan, tienen que cumplir, garantizando características que pueden ser usadas por motores de interpretación y también por editores.

# Beneficios del "strong typing"

- Nuestro array de productos no tiene ningún tipo definido, por lo que el editor no puede comprobar los errores mientras se escribe.
- Una de las formas de evitarlo, es definir una interfaz con todas las propiedades que el array deba tener, y declarar el array como de ese tipo de interfaz.
- Así que creamos un fichero separado "**producto.ts**" en el mismo directorio "**productos**", y definimos la interfaz de esta forma:



```
i.json x producto.ts x lista-productos.component.ts
export interface IProducto {
  "IdProducto": number,
  "Producto": string,
  "Codigo": string,
  "Disponible": string,
  "Descripcion": string,
  "Precio": number,
  "Valoracion": number,
  "imageUrl": string
}
```

# Beneficios del "strong typing"

- A continuación, en el componente, tenemos que realizar la importación de la interfaz, y redefinir los datos para que se moldeen según la interfaz:

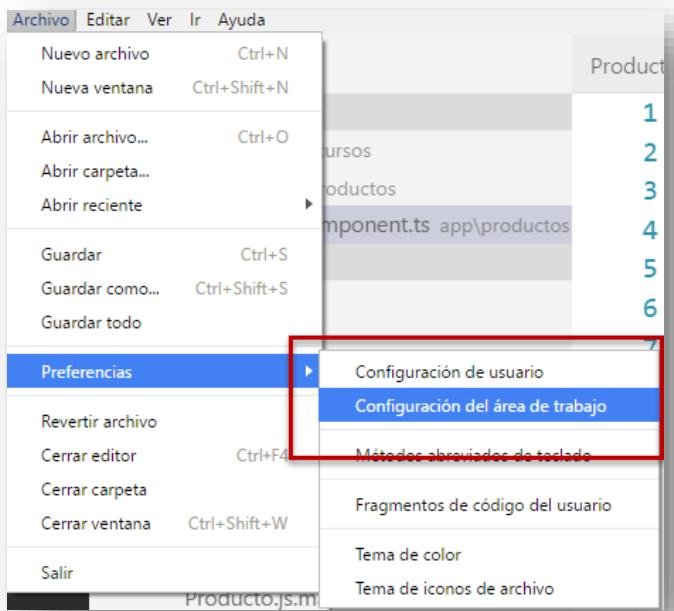
```
import { NgIf, NgFor } from '@angular/common';
import { IProducto } from './Producto';

@Component({
  selector: 'lista-productos',
  templateUrl: 'app/productos/lista-productos'
})
export class ListaProductosComponent {
  imageWidth: number = 50;
  imageMargin: number = 3;
  imageVisible: boolean = false;
  filtro: string = "cart";
  productos: IProducto[] = [
```

- Ahora, si cometemos un error, el editor nos avisa inmediatamente
- Lógicamente, los beneficios no terminan ahí...

# Configuración en detalle del editor

- Si queremos configurar el editor de manera que solo muestre los archivos que nos interesan durante el desarrollo (y no todos), para buscar más fácilmente, la solución está en el archivo "settings.json" de VStudio Code.
- Podemos añadir una entrada personalizada "**files.exclude**" indicando los directorios y/o extensiones que no queremos que muestre.



```
settings.json x
1 // Coloque su configuración en este archivo
2 {
3   "files.exclude": {
4     "**/.git": true,
5     "**/.svn": true,
6     "**/.hg": true,
7     "**/.DS_Store": true,
8     "**/*.js": true,
9     "**/*.map": true
10 }
11 }
```

# Encapsulación de estilos en componentes

- Utilizar el concepto de encapsulación para los estilos no es una idea nueva, pero su aplicación aquí sí lo es. Eso permite evitar "colisiones" de CSS, muy comunes en el desarrollo.
- Además, muchas veces las plantillas requieren de estilos únicos.
- Ponerlos en la plantilla está bien si es algo mínimo, pero, en cuanto crecen un poco, se hacen difíciles de mantener y más de reutilizar. También podemos hacerlo en una hoja externa, pero eso supone referenciar la hoja en **index.html**.
- Como ya hemos mencionado, los componentes admiten definiciones de estilos internas mediante las propiedades **styles** y **styleUrls** (que son arrays).

```
  styles: [ 'thead {color: #337AB7;}' ])}
```

- Es preferible crear una hoja propia para el componente. Así que definimos el archivo "**lista-productos.component.css**" dentro de la carpeta "**products**"

# Encapsulación de estilos en componentes

- Vamos a cambiar las cabeceras de columna para que tengan un color distinto y un tipo de letra algo más grande:



- A continuación declaramos la propiedad "styleUrls" en el componente:

```
@Component({  
    selector: 'lista-productos',  
    templateUrl: 'app/productos/lista-productos.component.html',  
    styleUrls: ['app/productos/lista-productos.component.css']  
})
```

A screenshot of a code editor showing an Angular component declaration. The 'styleUrls' property is highlighted with a blue box.

- Si todo va correcto, deberíamos observar los cambios inmediatamente.

# Encapsulación de estilos en componentes

- Observa que es posible indicar más de una hoja de estilo distinta, e incluso ubicar algunas hojas en una zona común (por ejemplo, para definiciones de tipo corporativo)
- En este caso tiene sentido mantener todo lo relacionado con el componente "*lista-productos*" en el mismo directorio.
- En principio, no hay ninguna restricción de estilo en lo que podemos incluir en un estilo de componente.

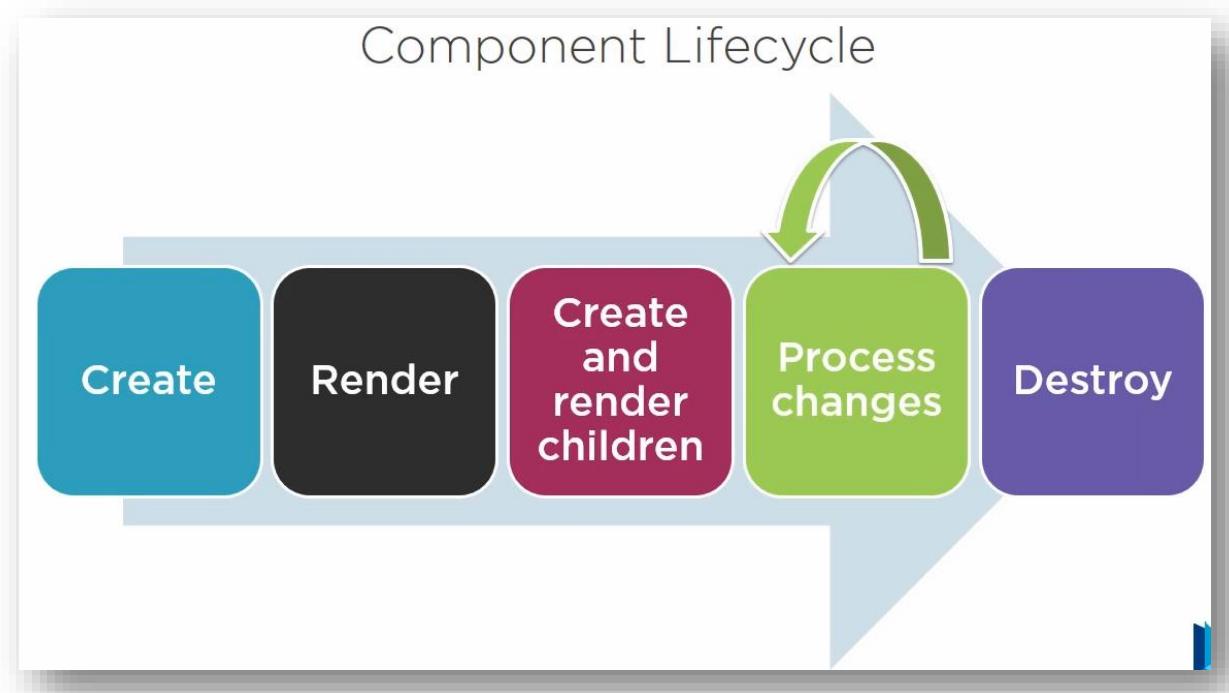
Filtrado por: cart

Ocultar imágenes

Producto	Codigo	Disponible	Precio	Valoracion
Leaf Rake	gdn-0011	March 19, 2016	€19.95	3

# Uso de recursos del ciclo de vida de un componente

- Los componentes tienen un "ciclo de vida" que son los pasos que realiza Angular 2 para crearlo y mantenerlo, hasta que es eliminado del DOM



# Uso de recursos del ciclo de vida de un componente

- El programador puede intervenir entre estas fases, e incluir "*lifecycle hooks*" para modificar el comportamiento predeterminado.
- Los 3 más utilizados son:
  - **onInit** : para la inicialización de componentes. Es paso perfecto para realizar la lectura de datos desde un servidor "*back-end*".
  - **onChanges**: para ejecutar código tras producirse un cambio en un elemento de entrada.
  - **onDestroy**: Permite realizar tareas de limpieza de memoria antes de destruir el componente.
- La implementación tiene lugar mediante implementación de interfaces de estos "hooks" que ya está definidos en Angular.
  - Esto requiere la referencia correspondiente al "hook" que vamos a utilizar
  - Y la implementación del método (similar a la de un manejador de evento).

# Uso de recursos del ciclo de vida de un componente

- Esquemáticamente, podríamos verlo en código según este gráfico:

```
import { Component, OnInit } from 'angular2/core';
export class ListaProductosComponent implements OnInit {
    ngOnInit(): void {
        console.log('In OnInit');
    }
}
```

- La implementación de interfaces no es imprescindible (no existen todavía en JavaScript), pero es una buena práctica.

# Uso de recursos del ciclo de vida de un componente

- De modo que nuestro código de implementación tendría las siguiente modificaciones

The screenshot illustrates the implementation of the `OnInit` lifecycle hook in an Angular component. On the left, the component's TypeScript code is shown:

```
import { Component, OnInit } from '@angular/core';
import { NgIf, NgFor } from '@angular/common';
import { IProducto } from './Producto';

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

On the right, the browser's developer tools are open to the 'Console' tab, showing the output of the `ngOnInit()` method:

```
Ejecutando OnInit
Angular 2 is running in the development mode.
```

A blue arrow points from the highlighted `ngOnInit()` code in the component's file to the corresponding log entry in the browser's console.

# Construcción de Pipes

- Un *Pipe* no es más que otro componente. Por lo tanto, una vez creado se pone en marcha de la misma forma (importación, referencias, etc.)
- Por tanto se construye mediante una clase decorada. Aunque el decorador en este caso es **@Pipe**.
- Se suelen utilizar las mismas convenciones de nomenclatura, tanto en la creación de la clase, como en los nombres de archivo.
- Empezaremos por crear un filtro para nuestra lista de productos. De modo que en la carpeta "**productos**" añadiremos un nuevo archivo "**lista-productos.pipe.ts**"

# Construcción de Pipes

- La funcionalidad asociada las *pipes* está incluida en los bloques *Pipe* y *PipeTransform* pertenecientes a **@angular.core**.

```
import { PipeTransform, Pipe } from "@angular/core";
import { IProducto } from './producto';

@Pipe({ name: 'filtroProductos'})
export class FiltroProductosPipe implements PipeTransform {
  transform(value: IProducto[], args: string[]): IProducto[] {
    let filtro: string = args[0] ? args[0].toLocaleLowerCase() : null;
    return filtro ? value.filter( (producto: IProducto) =>
      producto.Producto.toLocaleLowerCase().indexOf(filtro) != -1) : value;
  };
}
```

Nombre en la  
interfaz de usuario

- La propiedad ***name*** del decorador será la que usaremos en la interfaz de usuario (***filtroProductos***)

# Construcción de Pipes

- Para las referencias, lo fundamental es empezar por el componente base, que queda con estos cambios

The screenshot shows a code editor with several tabs at the top: 'ductos.pipe.ts' (selected), 'app.module.ts' (with an 'X'), 'lista-productos.component.ts', and 'lista-productos.component.html'. The 'app.module.ts' tab is currently inactive.

```
import { NgModule }      from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { FormsModule }   from '@angular/forms';
import { AppComponent }  from './app.component';
import { ListaProductosComponent } from './productos/lista-productos.component';
import { FiltroProductosPipe } from './productos/lista-productos.pipe';

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

The code in the 'app.module.ts' file imports the necessary modules and components. It then defines an @NgModule block with imports of BrowserModule and FormsModule, declarations of AppComponent and ListaProductosComponent, and bootstrap of AppComponent. The FiltroProductosPipe component is also declared within this block. The 'FiltroProductosPipe' line is highlighted with a blue rectangle, indicating it is the current focus of the discussion.

# Construcción de Pipes

- Y, en la interfaz de usuario, solo tenemos que utilizarla como cualquier otra *pipe*.

```
<tr *ngFor="let producto of productos | filtroProductos:filtro">
```

- Y como la palabra "cart" está asociada el miembro "*filtro*" del componente, ya veremos el filtro desde el inicio

## Demo de Gestión de Productos

The screenshot shows a user interface for managing products. At the top, there's a blue header bar with the text "Lista de productos". Below it, a search bar contains the text "cart". An arrow points from this text to the label "Filtrado por: cart" located below the search bar. Another arrow points from the "Ocultar imágenes" button to a table below. The table has columns: Producto, Código, Disponible, Precio, and Valoración. A single row is visible, showing "Garden Cart", "gdn-0023", "March 21, 2016", "€32.99", and "4.2".

Producto	Código	Disponible	Precio	Valoración
Garden Cart	gdn-0023	March 21, 2016	€32.99	4.2



## Pongámoslo en práctica

- Añade un pipe que permita filtrar la lista de tareas en función de dos parámetros: un campo y el valor del mismo

8

# RUTAS Y NAVEGACIÓN

# Sistema de navegación

- El sistema de navegación ha sido modificado varias veces hasta quedar definido en su versión final (>= RC5).
- La estructura final se apoya igualmente en el modelo SPA, pero el elemento que contiene las partes variables se denomina (por convención):

```
<router-outlet></router-outlet>
```

- Existen un objeto que controla el componente a visualizar dentro de ese elemento: app.routing.ts (gestor de URLs)
- En el caso de la navegación interna se utilizan bindings especiales designados con

```
[routerLink] = ["ruta a navegar"]
```

- Finalmente, también podemos navegar desde el código utilizando el objeto Router.

# Modelo de rutas

- Para configurar el sistema de rutas debemos de:
  - Trasladar la funcionalidad actual de app.component a otro componente, dejando solamente el sistema de rutas.
  - Crear el nuevo componente separado
  - Definir el elemento base en index.html
  - Registrar los proveedores de rutas.
  - Enlazar el proveedor de rutas con la aplicación.
- Por tanto la navegación se define en el sentido Path => Componente (navegación lógica) en lugar de utilizar el modelo Path => Recurso.html (Navegación física).
  - Este modelo es el típico de las aplicaciones MVC/SPA

# Cambios en la página maestra

- El primer paso lo tenemos que realizar en index.html:
  - Se debe de declarar qué ruta entendemos por ruta base, mediante el elemento `<base href="/">` en la cabecera del documento.
  - A continuación, modificamos ligeramente el elemento `<gestion-productos>`, para que esté empotrado en otro elemento (como un `<section> </section>`), que pueda servir de contenedor para el formato visual con clases Bootstrap
    - No obstante, ten presente que esto es solo por la parte visual, lo único realmente imprescindible aquí en lo funcional es el elemento `<base>`
  - Al final tendremos algo como:

```
<!-- 3. Muestra la aplicación -->
<body class="container-fluid">
  <section class="row">
    <gestion-productos class="col-sm-12 col-md-12 col-lg-12">
      Cargando...
    </gestion-productos>
  <section>
</body>
```

# Creación del menú y del "placeholder"

- El segundo paso consiste en modificar la plantilla de app.component, para que genere un menú con un par de opciones e incluya el elemento que va a ser el destino de las rutas: <router-outlet>

```
@Component({  
  selector: 'gestion-productos',  
  template: `  
    <h1>{{tituloPagina}}</h1>  
    <a class="btn btn-info" [routerLink] = "[ '' ]">Inicio</a>  
    <a class="btn btn-info" [routerLink] = "[ 'listado' ]">Lista Productos</a>  
    <br/><br/>  
    <router-outlet></router-outlet>  
  `,  
  providers: [ ServicioProductos ]  
})
```

- En este momento, hemos sustituido el elemento <listado-productos> anterior por el nuevo placeholder de rutas definido con ese nombre por convención dentro de Angular 2.
  - Más adelante entraremos en el significado de los atributos [routerLink]

# El componente de rutas

- El tercer paso, es crear otro componente (componente de bienvenida), de forma que podamos tener un par de ubicaciones distintas para navegar (ahora solo tenemos una).
- Así que, en otra carpeta (home) creamos un Welcome.component con sus dos partes: TypeScript y HTML.
- La parte que interesa, TypeScript puede ser como cualquier otro componente:



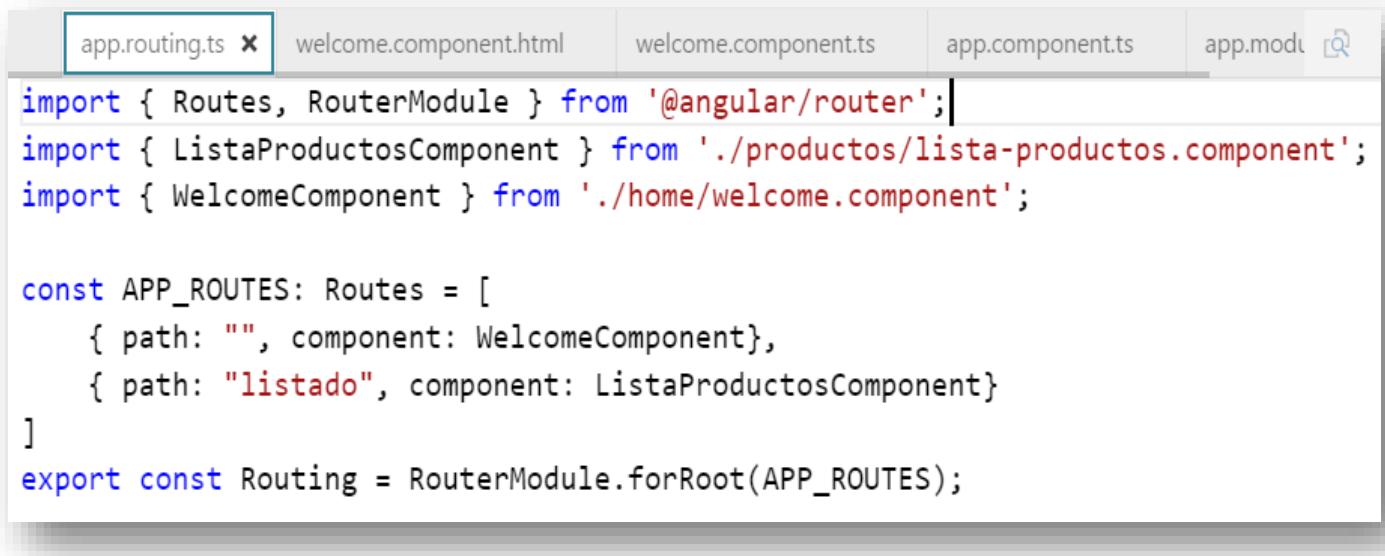
```
nl x app.routing.ts welcome.component.html welcome.component.ts x app  
import { Component } from '@angular/core';  
  
@Component({  
    templateUrl: 'app/home/welcome.component.html'  
})  
export class WelcomeComponent {  
    public pageTitle: string = 'Página de bienvenida';  
}
```

# El componente de rutas

- El cuarto paso, es crear el componente que se va a hacer cargo de las rutas (de todas las rutas en principio).
  - Como es global a la aplicación, lo declaramos en el mismo nivel que **app.module** y **app.component**.
  - El componente, debe de importar (para hacer referencia a ellos), todos los elementos que vayan a ser destino de una de sus rutas.
  - Además utiliza dos componentes separados del "core" de Angular (**Router** y **RouterModule**).
- El objetivo es crear una constante de tipo **Routes** (formato de array), que defina en sus elementos tantas parejas { **path** / **component** } como rutas queramos para nuestra aplicación.
  - Esta constante puede tener el nombre que queramos
- Por último esa constante debe ser expuesta al exterior para que el componente principal (**app.module**) pueda registrarla.

# El componente de rutas

- El código resultante no es muy complicado, solo hay que entender el propósito de cada elemento:



```
app.routing.ts x welcome.component.html welcome.component.ts app.component.ts app.modi 🔎

import { Routes, RouterModule } from '@angular/router';
import { ListaProductosComponent } from './productos/lista-productos.component';
import { WelcomeComponent } from './home/welcome.component';

const APP_ROUTES: Routes = [
  { path: "", component: WelcomeComponent},
  { path: "listado", component: ListaProductosComponent}
]
export const Routing = RouterModule.forRoot(APP_ROUTES);
```

- El método ***forRoot*** de ***RouterModules*** asocia las rutas definidas con la constante de tipo ***Routes*** que ahora podremos usar para el registro

# Registro de rutas

- Así que en el módulo principal añadiremos las referencias necesarias a este elemento:
- En la parte de declaraciones, ahora deben aparecer las referencias a **Router** y al nuevo componente **WelcomeComponent**.

The screenshot shows the `app.module.ts` file in an IDE. The code imports various Angular modules and components. Two specific imports are highlighted with a teal border: `import { Routing } from './app.routing';` and `import { WelcomeComponent } from './home/welcome.component';`. A large blue arrow points from the word `Routing` in the first highlighted line down to the second highlighted line, indicating the relationship between the Router module and the component it is being used to define.

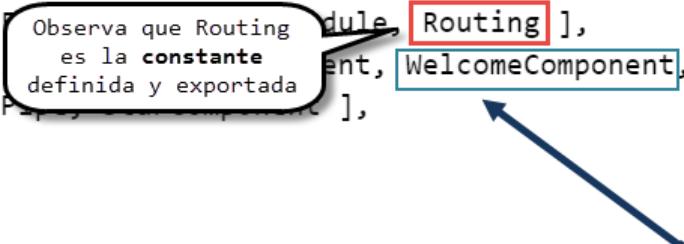
```
app.routing.ts app.component.ts app.module.ts x
import { NgModule }      from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { FormsModule } from '@angular/forms';
import { HttpClientModule } from '@angular/http';
import { Routing } from './app.routing';
import 'rxjs/Rx';

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

# Registro de rutas

- En el decorador @NgModule, incluimos el nuevo componente en el apartado ***declarations***, y la referencia al **Router** dentro de los **imports** (componentes importados).

```
@NgModule({  
  imports:      [ BrowserModule, RouterModule ],  
  declarations: [ AppComponent, LoginComponent, WelcomeComponent, FiltroProductosComponent ],  
  bootstrap:    [ AppComponent ],  
})  
export class AppModule { }
```



- De esa forma las rutas definidas en la constante **Routing** quedan disponibles para cualquier elemento manejado por el módulo principal.

# Resultados finales

- Además, si nos fijamos en la definición de las rutas, estábamos usando como atributo **path** las cadenas "" y "**listado**" respectivamente.
- Se entiende que "" (cadena vacía) hace referencia a la raíz del sitio, y que tenemos asociada con **WelcomeComponent**.
- Y "**listado**" es un fragmento a resolver que hemos asociado **con un componente** (en este caso, con **ListadoComponent**).
- Por tanto, si nuestro sitio está configurado por defecto para comenzar por la URL **localhost:3000**, deberemos ver esa salida de forma inicial



# Enlaces a rutas en la vista

- Pero, para poder ver el listado de productos necesitaríamos teclear manualmente en la URL el fragmento "*listados*".
- Por esa razón usamos los atributos **[routerLink]**, que asocian el valor indicado con las rutas definidas.

```
@Component({  
    selector: 'gestion-prod',  
    template: `/  
        <h1>{{tituloP}}</h1>  
        <a class="btn btn-info" [routerLink] = "[ '' ]">Inicio</a>  
        <a class="btn btn-info" [routerLink] = "[ 'listado' ]">Lista Productos</a>  
        <br/><br/>  
        <router-outlet></router-outlet>  
    `,  
    providers: [ ServicioProductos ]  
})
```

The code snippet shows a component definition for 'gestion-prod'. It includes a template with an h1 tag and two anchor tags. The second anchor tag uses the [routerLink] attribute to link to the 'listado' route. A callout box highlights the APP\_ROUTES constant and the routerLink attribute. Arrows point from the highlighted text to the corresponding parts in the code.

- Con esta asignación al seleccionar el enlace "listado" navegaremos a nuestro listado básico.



## Pongámoslo en práctica

- Crea una rutas diferentes para las vistas de tareas y proyectos.

# Rutas con path params

- Podemos generar para las rutas patrones que impliquen parámetros, de tal manera que todas las rutas que cumplan con el patrón sean servidas por el mismo componente.
- El formato que adoptará será el siguiente:

```
{ path: 'ruta/:param1/subruta/:param2', component: Componente }
```

- Por ejemplo:

```
{ path: 'producto/:id', component: DetalleProductoComponente }
```

# Rutas con path params

- Podemos leer en el componente destino la información de parámetros

```
this.route.params.subscribe(params => {  
  let id = params['id'];  
  ...  
});
```

- Para navegar por código:

```
this.router.navigate(['/producto', producto.id]);
```



## Pongámoslo en práctica

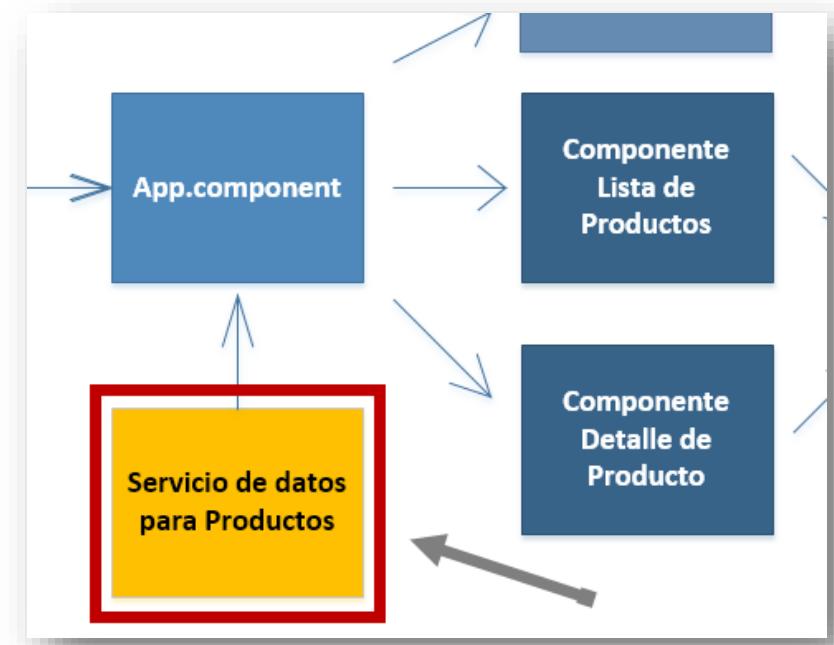
- Crea un componente que permita ver el detalle de una tarea en una subruta.

9

# CREACIÓN Y USO DE SERVICIOS

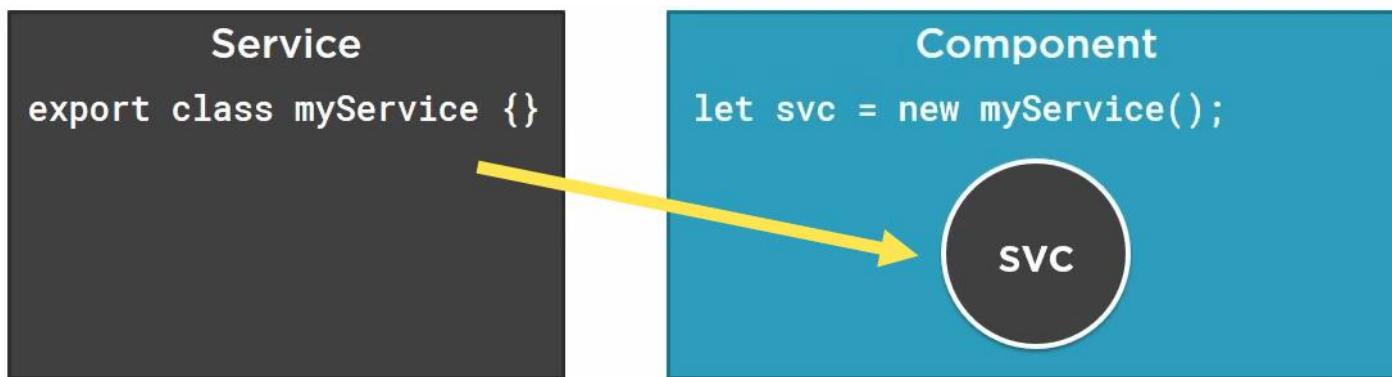
# Objetivos del módulo

- En éste módulo nos centraremos en la construcción de servicios, y de cara a nuestra aplicación de ejemplo, construiremos el servicio que proporciona los productos.
- De esa forma los datos serán suministrados por el servicio y podrán usarse desde distintas partes de la aplicación.



# Modos operativos

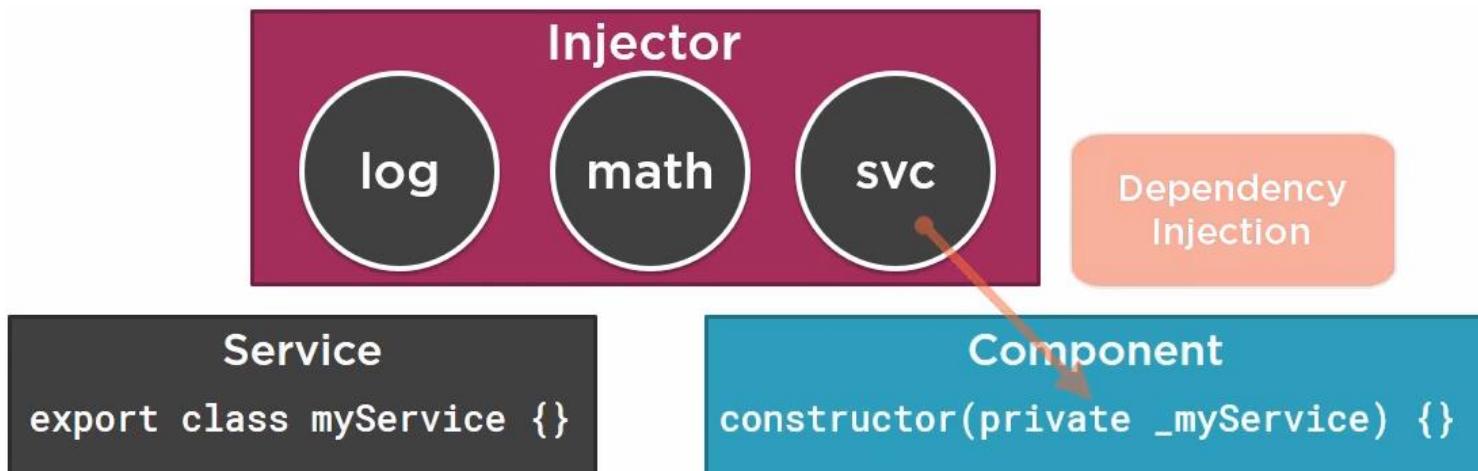
- Un servicio en Angular 2, es –por supuesto- una clase, declarada con el prefijo **export**, que aporta una cierta funcionalidad a otros componentes.
- Los clientes potenciales de este servicio, pueden declararlo de dos formas distintas:
  - Como una instancia (local a ese componente) (Ver diagrama)



- De esa forma, no pueden compartirse no datos ni otros recursos

# Modos operativos

- La forma preferida de trabajar con servicios es mediante su registro a nivel de aplicación.
- En este caso podremos solicitar el servicio por inyección de dependencia en cualquier componente y podremos compartir datos y recursos.
- Equivale a contar con un zona común de recursos para la aplicación, manejada por un "**injector**" que suministra instancias "**singleton**" de esos servicios a esos clientes.



# Codificación del servicio

- Observa que la inyección de dependencia funciona de manera similar a como lo hace en Angular 1.x: declara un objeto del tipo de un servicio, y el sistema de inyección es el encargado de suministrarlo.
- Al hacerlo en el constructor, la funcionalidad del servicio estará presente en el ámbito de la clase.
- Cuando creamos un servicio debemos marcarlo con el decorador que permite su manejo mediante inyección: **@Injectable**.
  - (Estrictamente, no es imprescindible marcarlo así, salvo que –el mismo– tenga otras dependencias de inyección, pero se considera una buena práctica hacerlo de esta forma.)
  - Además, todo lo declarado en el servicio es considerado com **public** de forma predeterminada.
  - Si queremos proteger esa información deberemos de declararlo como **private** o **protected**.
- Al crearlo, lógicamente, tendremos que importar todos los elementos que necesita, igual que hemos visto en módulos anteriores.
- Si el servicio no declara propiedades (**property**) se supone que no va a compartir información con otros elementos.

# Codificación del servicio

- Por lo tanto , crearemos solamente un método que devuelva como valor de retorno los datos que ahora estaban asignados a la propiedad **productos** de nuestro componente **listaproductos.component**.

```
import { Injectable } from "@angular/core";
import { IProducto } from "./producto";

@Injectable()
export class ServicioProductos {
  leerProductos(): IProducto[] {
    return [
      {
        "IdProducto": 1,
        "Producto": "Leaf Rake",
        "Codigo": "GDN-0011",
        "Disponible": "March 19, 2016",
        "Descripcion": "Escobilla para desatascar e"
    ]
  }
}
```

# Registro de un servicio

- Para registrar un servicio, tenemos que registrar un ***Provider***.
- Un ***provider*** es, simplemente, un código que crea o devuelve un servicio.
  - Normalmente, la propia clase
- Y para registrar un ***provider*** debemos definirlo como parte de los metadatos del componente.
- De esta forma, el ***injector*** puede manejarlo desde el componente donde se declara y también para cualquiera de sus descendientes jerárquicos.
  - De ahí que sea importante el sitio donde se registra y eso dependa siempre de la funcionalidad que queramos dar al servicio según la arquitectura de la aplicación.

# Registro de un servicio

- Con esas indicaciones, el código resultante de la clase ***lista-productos.component.ts*** quedará de la siguiente forma:

```
s.service.ts      app.component.ts ✘    lista-productos.component.ts
import { Component } from '@angular/core';
import { ServicioProductos } from './productos/productos.service';

@Component({
  selector: 'gestion-productos',
  template: `
    <h1>{{tituloPagina}}</h1>
    <lista-productos></lista-productos>`,
  providers: [ ServicioProductos ]
})
export class AppComponent {
  tituloPagina: string = "Demo de Gestión de Productos";
}
```



# Utilización del un servicio registrado

- Para utilizar el servicio, debemos declararlo en los metadatos del componente que lo usa: importarlo, y definir el servicio por inyección en el constructor de la clase correspondiente.
- Para ponerlo en marcha una vez definido, podemos utilizar el evento ***ngOnInit***, que antes solo usábamos para mostrar algo por consola.
- Finalmente, borraremos los datos "empotrados" en la propiedad ***productos*** y vaciaremos otras propiedades del mismo tipo.
- En resumen, el conjunto de cambios es el siguiente:
- Importación del servicio de productos:

```
import { Component, OnInit } from '@angular/core';
import { NgIf, NgFor } from '@angular/common';
import { IProducto } from './Producto';
import { ServicioProductos } from './productos.service';
```

# Utilización de un servicio registrado

- Borrado de datos "hard-coded" (empotrados) y declaración del constructor que requiere el servicio por inyección de dependencia.

```
export class ListaProductosComponent implements OnInit {  
    imageWidth: number = 50;  
    imageMargin: number = 3;  
    imageVisible: boolean = false;  
    filtro: string='';  
    productos: IProducto[];  
  
    constructor(private _servicioProductos: ServicioProductos){}  
}
```

- Reescritura del evento "*ngOnInit*" para leer los datos:

```
ngOnInit() {  
    // console.log("Ejecutando OnInit");  
    this.productos = this._servicioProductos.leerProductos();  
}
```

# Salida final

- Finalmente, como hemos puesto un registro más de datos en el servicio, veremos un elemento más en la salida
- (Además hemos limpiado la asignación inicial a "cart" para que no tenga filtros, y podamos ver los datos disponibles).

## Demo de Gestión de Productos

Lista de productos					
Filtrar por:					
Ocultar imágenes	Producto	Código	Disponible	Precio	Valoración
	Leaf Rake	gdn-0011	March 19, 2016	€19.95	★★★
	Garden Cart	gdn-0023	March 21, 2016	€32.99	★★★★
	Hammer	tbx-0048	May 21, 2016	€8.90	★★★★★



## Pongámoslo en práctica

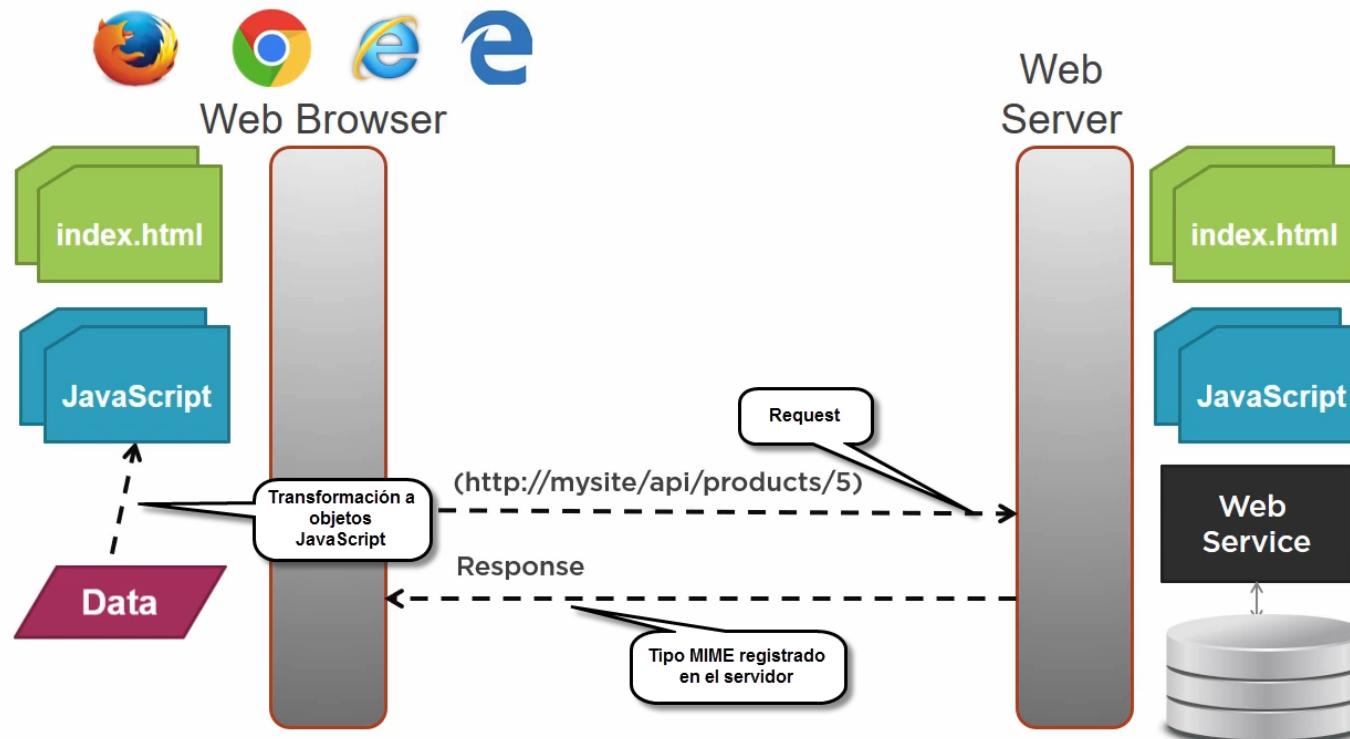
- Crea un servicio para las tareas que entregue los datos de las tareas y la funcionalidad de borrar tareas.
- Haz lo propio para los proyectos.

10

## ACCESO A APIs DE SERVIDOR

# Objetivos del módulo

- Este módulo trata sobre los mecanismos de acceso a datos en Angular 2
- Especialmente, los relativos al uso del protocolo HTTP y las colecciones ***observable***.
- Recordemos el diagrama básico de comunicación cliente/servidor



# Observables

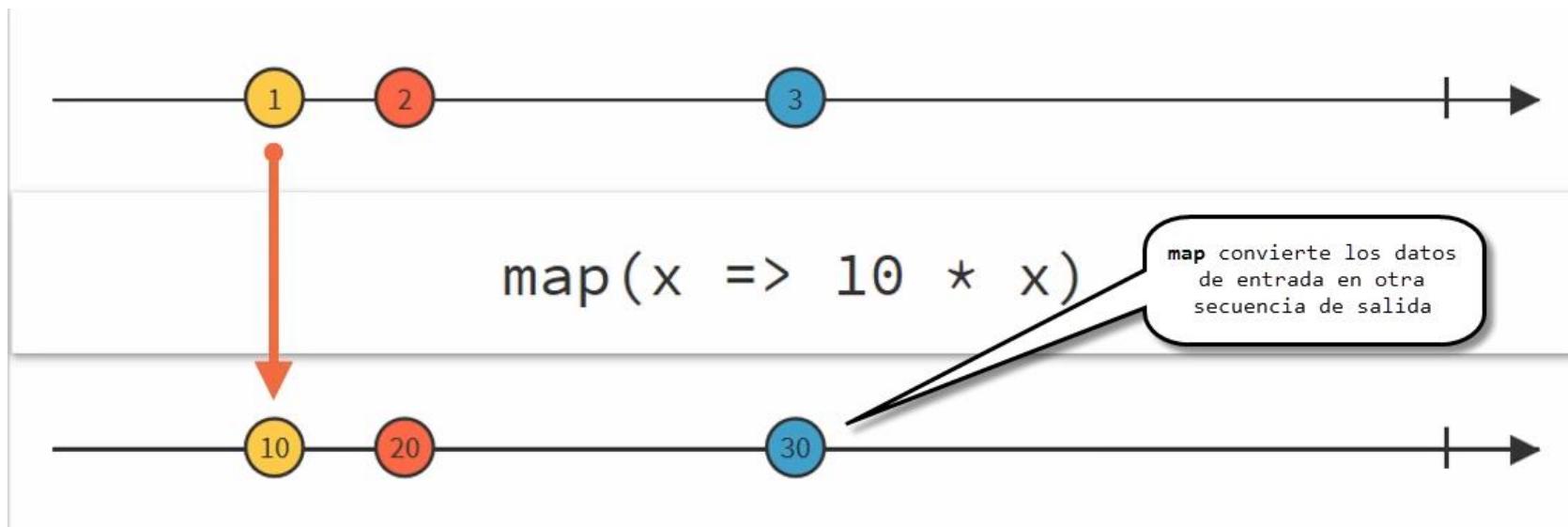
- Entendemos por **observable** un array que llega a un destino en bloques (**marbles**) de forma asíncrona.



- Son fundamentales en peticiones de servicios para unir de forma coherente datos que llegan de manera muy dispar.
- Son una característica propuesta para ES2016, pero para poderlo utilizar ahora, Angular usa la librería **RxJS (Reactive Extensions)**.
- Angular 2 también los utiliza en sus sistema de eventos.

# Observables

- El funcionamiento de los observables queda explicado en el siguiente diagrama, donde vemos el uso de la sentencia **map( x => x \* 10)**, para convertir una secuencia observable en otra distinta.



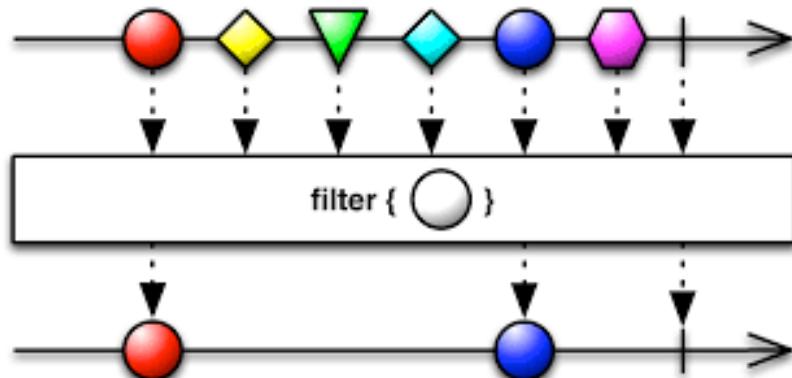
- Existe una cierta similitud con las **Promises**

# Observables: diferencias respecto a las *Promises*

- Hay diferencias notables entre ambos:
  
- **Promises**
  - Devuelven un único valor
  - No son cancelables
  - También se pueden usar en Angular 2 (existe algún ejemplo de uso en la documentación oficial).
  
- **Observables**
  - Devuelven valores múltiples en el tiempo.
  - Pueden cancelarse
  - Soportan operadores modernos como **map**, **reduce**, **filter**, **retry**, etc.
  - Utilizan una arquitectura parecida a los Web Sockets.

# Observables: Configuración

- Para utilizar observables con HTTP se requieren algunos pasos previos:
  - Incluir el **script** de Http para Angular 2
  - Registrar HTTP\_PROVIDERS
  - Importar Reactive Extensiones (RxJS)
- Esto nos permite utilizar otro conjunto de recursos como los filtros:



# Observables: Configuración

- Las secuencias de datos pueden tomar diversas formas:
  - **Streams** provenientes de un servidor de back-end o servicio web.
  - Un conjunto de notificaciones de sistema
  - Series de eventos, tales como las entradas del usuario
- Nuestro código se puede suscribir para recibir esas notificaciones asíncronas, según van llegando los datos.
- De esa forma puede "reaccionar" cuando llegan los datos, o cuando se le notifica un error.
- Las "*Reactive extensions*" proporcionan, además, otras funcionalidades.

# Observables: Configuración

- La forma en que se expone la funcionalidad es a través de una constante, que aglutina unos cuantos servicios relacionados con HTTP.
- Para disponer de esa funcionalidad en diversas partes de la aplicación, importamos el módulo **HttpModule** en el componente principal, y también todo lo incluido en las "**reactive extensions**".
  - ¡Ojo! En versiones anteriores a la final, se utilizaba una constante **HTTP\_PROVIDERS**, que ya no funciona en la versión final).
- En nuestro caso, deberemos de importar esos elementos en **app.module.ts**, y declarar **HttpModule** en la lista de **imports**:

```
import { HttpModule } from '@angular/http';
import 'rxjs/Rx';
```

```
@NgModule({
  imports:      [ BrowserModule, FormsModule, HttpModule ],
  declarations: [ AppComponent, ListaProductosComponent,
                 FiltroProductosPipe, StarComponent ],
  bootstrap:    [ AppComponent ],
})
```

# Observables: Programación

- De esta forma se expone su funcionalidad en todos los módulos dependientes.
- Por otro lado, nuestro componente no sufre modificaciones, solamente cambiaremos el servicio para que lea los datos y le añadimos una rutina de tratamiento de errores para ayuda en la depuración.
- Así pues, el servicio queda de la siguiente forma:
  - En la parte de declaraciones, ahora declaramos los componentes **Http** y **Response** del módulo http y **Observable** de las RX:

```
import { Injectable } from '@angular/core';
import { IProducto } from './producto';
import { Http, Response } from '@angular/http';
import { Observable } from 'rxjs/Observable';
```

# Observables: Programación

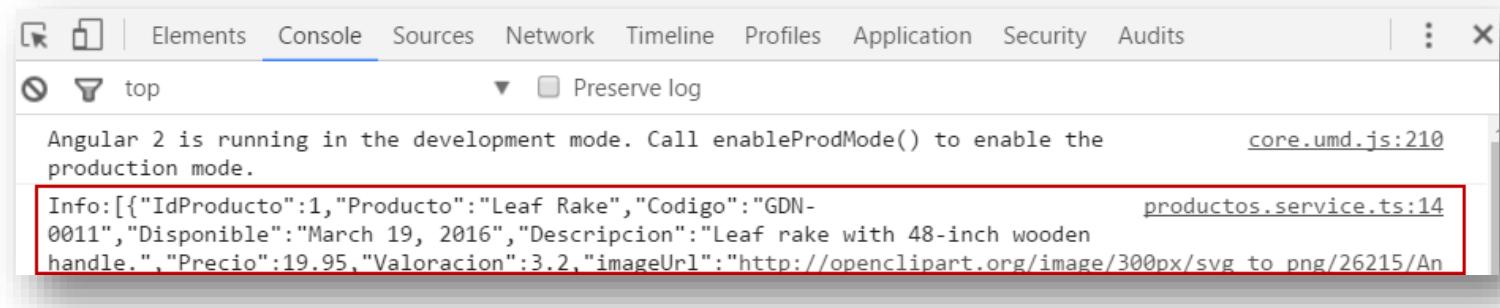
- La clase es la que más modificaciones tendrá.
- Por un lado declaramos una variable local para la Url de los productos.
- Además, tenemos que declarar (por inyección de dependencia) el servicio Http, y para la lectura eliminamos los datos y los sustituimos por la llamada correspondiente:

```
@Injectable()
export class ServicioProductos {
    private _productosUrl = 'app/productos/Productos.json';
    constructor(private _http: Http){}

    leerProductos(): Observable<IProducto[]> {
        return this._http.get(this._productosUrl).
            map( (response: Response) => <IProducto[]>response.json() ).
            do(datos => console.log('Info: ' + JSON.stringify(datos))).
            catch(this.manejarError);
    }
}
```

# Observables: Programación

- Observa que la llamada devuelve un **Observable** genérico, y – cuando los datos se reciben- se recogen en objeto **response** mediante **map**, después de haber convertido el "stream" a **json**.
- Además utilizamos el método **do()** para realizar otra labor adicional, presentar los datos por consola, de forma que los podamos ver allí directamente, además de en la IU:



- Finalmente, creamos una rutina de tratamiento de error (**manejarError**), que presenta los errores potenciales por consola y también lanza una excepción.

# Observables: Programación

- La rutina de error es sencilla:

```
private manejarError(error: Response) {
    console.error(error);
    return Observable.throw(error.json().error || "Error del servidor");
}
```

- Si todo ha ido bien, deberemos de ver finalmente los cinco productos del fichero *json* en el listado final.

Producto	Código	Disponible	Precio	Valoración
Leaf Rake	gdn-0011	March 19, 2016	€19.95	★★★
Garden Cart	gdn-0023	March 18, 2016	€32.99	★★★★
Hammer	tbx-0048	May 21, 2016	€8.90	★★★★★
Saw	tbx-0022	May 15, 2016	€11.55	★★★★
Video Game Controller	gmg-0042	October 15, 2015	€35.95	★★★★★



## Pongámoslo en práctica

- Has que los servicios accedan a una API para traer las tareas y proyectos.



## Pongámoslo en práctica

- Haz que el servicio de proyectos se comporte como una store.



## BananaTube: Decisión en equipo

- Discute en equipo la conveniencia de usar Angular 2 para BananaTube
- Tomad nota de las fortalezas y debilidades que hayas observado en el framework, así como, la dificultad de incorporarlo en el proyecto.



[...]**netmind**

WeKnowIT

Barcelona

C. Almogàvers, 123  
08018 Barcelona  
Tel. 93 304.17.20  
Fax. 93 304.17.22

Madrid

Plaza Carlos Trías Bertrán, 7  
28020 Madrid  
Tel. 91 442.77.03  
Fax. 91 442.77.07

[www.netmind.es](http://www.netmind.es)



GOBIERNO  
DE ESPAÑA

MINISTERIO  
DE ENERGÍA, TURISMO  
Y AGENDA DIGITAL

**red.es**



ESTRATEGIA DE  
EMPRENDIMIENTO Y  
EMPLEO JUVENIL  
*garantía juvenil*



Agenda Digital para España



**UNIÓN EUROPEA**

Fondo Social Europeo  
*"El FSE invierte en tu futuro"*