

Semana 2

Curso de Angular



Semana 2
Data binding. Pipes.
Estilo y ciclo de vida de
los componentes.

Desarrollo de aplicaciones web con Angular
Cefire 2017/2018
Autor: Arturo Bernal Mayordomo

Index

Vinculación de datos (Data binding).....	3
Vincular atributos (Property binding).....	3
ngStyle and ngClass.....	4
Vincular eventos (Event binding).....	5
Vinculación bidireccional → [(ngModel)].....	6
Filtros (Pipes).....	8
Creando nuestros filtros.....	9
Más sobre componentes.....	11
Encapsulación de estilos en componentes.....	11
Ciclo de vida de los componentes.....	12

Vinculación de datos (Data binding)

Vincular atributos (Property binding)

Vincular atributos (Property binding) sigue el mismo concepto que la `{{interpolación}}`, pero aplicado a los atributos de elementos HTML. Se vinculará la propiedad del componente actual (puede ser un valor calculado o el resultado de una llamada a una función) a un atributo HTML cualquiera (`src`, `title`, ...).

Para establecer un atributo vinculado al componente (su valor se calcula y se actualizará en tiempo real), debemos rodear el nombre del atributo entre corchetes (ejemplo: `[src]`). El valor será el atributo de la clase del componente o la llamada al método que nos devuelva el valor a asignar en cada instante de tiempo.

Siguiendo con el ejemplo de la primera parte, vamos a mostrar la imagen de cada producto en la primera columna de la tabla. Las imágenes, al igual que otros archivos multimedia o de texto auxiliares se deben situar en la carpeta **src/assets** del proyecto, que será exportada tal cual al compilarlo.

```
<table class="table table-striped">
  <thead>
    <tr>
      <th>{{headers.image}}</th><th>{{headers.desc}}</th>
      <th>{{headers.price}}</th><th>{{headers.avail}}</th>
    </tr>
  </thead>
  <tbody>
    <tr *ngFor="let product of products">
      <td><img [src]="product.imageUrl" alt=""></td>
      <td>{{product.desc}}</td>
      <td>{{product.price}}</td>
      <td>{{product.avail}}</td>
    </tr>
  </tbody>
</table>
```

De hecho, podríamos usar la sintaxis de interpolación para vincular atributos. Sin embargo, la sintaxis de los corchetes es la recomendada.

```
<img [src]="product.imageUrl" alt=""> → Mejor!
<img src={{product.imageUrl}} alt="">
```



Al igual que los atributos HTML, podemos vincular estilos CSS (**style binding**) a valores calculados. Para ello, se usa el atributo `style`, seguido del nombre de la propiedad CSS (en formato camelCase), y si esta lo requiere, el tipo de unidad de medida. Por ejemplo, si queremos establecer una altura en píxeles en base a una propiedad llamada `imageHeight` → `[style.height.px]="imageHeight"`.

Lo recomendable es siempre que sea un valor fijo que no cambie a lo largo del tiempo, usar los archivos de estilo asociados a la plantilla del componente → **src/app/product-list/product-list.component.css**:

```
td {
  vertical-align: middle;
}

td:first-child img {
  height: 40px;
}
```

Welcome to Angular Products!

Mi lista de productos			
Imagen	Producto	Precio	Disponible
	SSD hard drive	75	Mon Oct 03 2016 02:00:00 GMT+0200 (CEST)
	LGA1151 Motherboard	96.95	Thu Sep 15 2016 02:00:00 GMT+0200 (CEST)

ngStyle and ngClass

Lo recomendado si quieres cambiar dinámicamente el estilo de un elemento (sobre todo si afecta a varios atributos), es usar las directivas **ngStyle** y **ngClass**.

La directiva **ngStyle** recibe un objeto como valor. Este objeto tiene como nombres de atributos las propiedades CSS a modificar, y como valores, la propiedad, o el método del componente que establecerá su valor CSS. El objeto se puede asignar directamente en el HTML o dentro del componente.

En el siguiente ejemplo, si la propiedad (isEven: boolean) es true, el color de fondo de la celda será rojo, y **cuando** cambie (false), pasará a ser verde.

```
<td [ngStyle]="{'background-color': isEven?'red':'green'}">...</td>
```

Para especificar unidades de medida (px, em, mm, ...), lo añadimos como sufijo al nombre de la propiedad CSS → propiedad.unidades:

```
<td [ngStyle]="{'width.px': width}">...</td> → width: number
```

Como hemos comentado antes, se puede crear el objeto con las propiedades dentro del componente (más limpio) y vincularlo desde la plantilla:

```
<td [ngStyle]="styleObject">...</td>
```

La directiva **ngClass** es similar aunque más simple. Simplemente asigna una clase CSS, o no, en función de un valor booleano.

```
<td [ngClass]="{'even': isEven, 'last': isLast}"> → isEven y isLast: boolean
```

Puedes vincular más de una clase al valor de una propiedad:

```
<td [ngClass]="{'even': isEven, 'last active': isLast}">
```

Como ngStyle, se puede crear el objeto en la clase del componente y referenciarlo en la plantilla:

```
<td [ngClass]="classObject">
```

```
@Component({
  ...
})
export class SomeComponent {
  ...
  classObject = {
```

```

    'class1': true,
    'class2': false
  }
  ...
}

```

Vincular eventos (Event binding)

Lo “opuesto” de la vinculación de atributos podría ser la **vinculación de eventos**. Lo de opuesto viene porque cuando vinculamos un atributo, la comunicación va **unidireccionalmente** desde el componente a la plantilla, mientras que al vincular un evento, estamos yendo desde la plantilla al componente (llamada a un método).

Vinculamos un evento poniendo el nombre del mismo entre paréntesis como **(click)**, **(mouseenter)**, **(keypress)**, etc. En el ejemplo, vamos a crear un botón, y al hacer clic sobre él, las imágenes se mostrarán u ocultarán.

```

<thead>
  <tr>
    <th>
      <button class="btn btn-sm"
        [ngClass]="{'btn-danger': showImage, 'btn-primary': !showImage}"
        (click)="toggleImage()">
        {{showImage?'Ocultar':'Mostrar'}} imagen
      </button>
    </th>
    <th>{{headers.desc}}</th>
    <th>{{headers.price}}</th>
    <th>{{headers.avail}}</th>
  </tr>
</thead>
<tbody>
  <tr *ngFor="let product of products">
    <td>
      <img [src]="product.imageUrl" *ngIf="showImage" alt=""
        [title]="product.desc">
    </td>
    <td>{{product.desc}}</td>
    <td>{{product.price}}</td>
    <td>{{product.avail}}</td>
  </tr>
</tbody>

```

Se puede observar como se usa la directiva **[ngClass]**, en este caso para que el botón se muestre de un color u otro dependiendo de si las imágenes son visibles o no . Al hacer clic sobre el botón, se llama al método **toggleImage()** del componente.



```

showImage = true;

...

toggleImage() {
  this.showImage = !this.showImage;
}

```

Ocultar imagen	Producto	Precio	Disponible
	SSD hard drive	75	Mon Oct 03 2016 02:00:00 GMT+0200 (CEST)
	LGA1151 Motherboard	96.95	Thu Sep 15 2016 02:00:00 GMT+0200 (CEST)

Mostrar imagen	Producto	Precio	Disponible
	SSD hard drive	75	Mon Oct 03 2016 02:00:00 GMT+0200 (CEST)
	LGA1151 Motherboard	96.95	Thu Sep 15 2016 02:00:00 GMT+0200 (CEST)

Vinculación bidireccional → [(ngModel)]

Hemos visto como vincular unidireccionalmente los valores de atributos HTML (componente → plantilla) y los eventos generados (plantilla → componente). Existe también un tipo de vinculación bidireccional a través de la directiva **ngModel**, utilizada generalmente con elementos de tipo <input>.

Mediante esta directiva, que se utiliza combinando la sintaxis de paréntesis y corchetes → [(ngModel)], se vincula el valor del campo del formulario a una propiedad del componente, de tal forma que si cambiamos el valor en la propiedad, se haría visible ese cambio en el campo y viceversa, si cambiamos el valor del campo, cambia el de la propiedad.

Lo primero que hay que tener en cuenta, es que la directiva **ngModel**, forma parte del módulo de Angular **FormsModule**, donde se ubica todo lo relacionado con la gestión de formularios. Para poder usarla, debemos importar dicho módulo en el módulo de nuestra aplicación:

```
import { FormsModule } from '@angular/forms';
...

@NgModule({
  ...
  imports: [
    BrowserModule,
    FormsModule
  ],
  ...
})
export class AppModule { }
```

Acto seguido, vamos a declarar la propiedad que vincularemos al campo del formulario inicializándola a cadena vacía.

```
export class ProductListComponent implements OnInit {
  ...
  filterSearch: string = ''; // Podríamos poner un valor por defecto
  ...
}
```

A continuación, vamos a crear un formulario, justo antes de la tabla de productos con un campo de texto, que usaremos en un futuro para filtrar los productos por el nombre. Además, vamos a mostrar un texto al lado con el valor actual de la propiedad (filterSearch), y se podrá observar como cambia al mismo tiempo que cambiamos el valor del campo de texto.

```
...
<div class="card-block">
  <form class="form">
    <div class="form-group row">
      <label class="col-form-label col-sm-2 text-sm-right"
        for="filterDesc">Filtro:</label>
      <div class="col-sm-5">
        <input type="text" [(ngModel)]="filterSearch" class="form-control"
          name="filterDesc" id="filterDesc" placeholder="Filter...">
      </div>
      <label class="col-form-label col-sm-5">
        Filtrando por: {{filterSearch}}</label>
      </div>
    </form>
  ...
</div>
...
```

Mi lista de productos

Filtro:

Hola



Filtrando por: Hola

Filtros (Pipes)

Los filtros o **pipes**, se combinan con la interpolación para transformar la información antes de mostrarla. Angular provee algunos [filtros predefinidos](#) para operaciones con strings, arrays, fechas, JSON, moneda, internacionalización, etc.

```
<tr *ngFor="let product of products">
  <td>
    <img [src]="product.imageUrl" *ngIf="showImage" alt=""
        [title]="product.desc | uppercase">
    </td>
    <td>{{product.desc}}</td>
    <td>{{product.price | currency}}</td>
    <td>{{product.avail | date}}</td>
</tr>
```



El filtro **uppercase**, convierte una cadena a mayúsculas antes de mostrarla. También existen los filtros **currency** para indicar que estamos ante un precio, y **date** para convertir una fecha a un formato más legible.

	SSD hard drive	\$75.00	Oct 3, 2016
	LGA1151 Motherboard	\$96.95	Sep 15, 2016

Algunos filtros admiten parámetros que les podemos pasar separados por el carácter dos puntos ':'. Por ejemplo, vamos a indicar al filtro **currency** que el precio está establecido en euros (**EUR**), y que debe mostrar el símbolo de la moneda (**symbol** → €). [Más información](#).

También indicaremos al filtro de la fecha el formato de la misma. En este caso dd/mm/yyyy. [Más información](#).

```
<tr *ngFor="let product of products">
  <td>
    <img [src]="product.imageUrl" *ngIf="showImage" alt=""
        [title]="product.desc | uppercase">
    </td>
    <td>{{ product.desc }}</td>
    <td>{{ product.price | currency:'EUR':'symbol' }}</td>
    <td>{{ product.avail | date:'dd/MM/y' }}</td>
</tr>
```

Ocultar imagen	Producto	Precio	Disponible
	SSD hard drive	€75.00	03/10/2016
	LGA1151 Motherboard	€96.95	15/09/2016

Creando nuestros filtros

Como vimos antes, los filtros (pipes) se usan para transformar una entrada de datos antes de mostrarlos. Hay algunos filtros predefinidos en Angular para trabajar con strings, fechas, precios, etc.

También podemos crear nuestros filtros personalizados. Por ejemplo, vamos a crear un filtro que transforma un array de productos (IProduct), filtrando todos los que contienen un cierto texto en el nombre.

Creamos el directorio pipes en **src/app** y ejecutamos:

```
mkdir pipes
cd pipes
ng g pipe product-filter
```

Recuerda que debemos usar la sintaxis **snake-case** para lo que creamos con Angular CLI. La herramienta adaptará el nombre dependiendo del contexto. En este caso, el filtro se llamará **productFilter** y su clase **ProductFilterPipe**.

Tal como sucede con los componentes, los filtros también se registran automáticamente en el array **declarations** del módulo de la aplicación:

```
@NgModule({
  declarations: [
    AppComponent,
    ProductListComponent,
    ProductFilterPipe
  ],
  ...
})
```

Así es como se ha creado la clase del filtro en **product-filter.pipe.ts**:

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

@Pipe({
  name: 'productFilter'
})
export class ProductFilterPipe implements PipeTransform {
  transform(value: any, args?: any): any {
    return null;
  }
}
```

Como puedes observar, es similar a la clase de un componente, pero usa el decorador **@Pipe**, para que Angular sepa que se trata de un filtro. El nombre, **productFilter**, es el que usaremos en la plantilla para aplicar el filtro.

Esta clase implementa la interfaz **PipeTransform**, que básicamente obliga a implementar el método **transform** que será el encargado de aplicar el filtro. Gracias a que estamos trabajando con TypeScript, podemos limitar el tipo de datos de entrada (parámetro **value**) para que sea un array de productos, y el del resto de parámetros. En este caso sólo tendremos un parámetro (podemos añadir los que queramos), llamado **filterBy** que será un string para filtrar productos en base a su nombre.

Este método devolverá un array de productos ya filtrados.

```
import { Pipe, PipeTransform } from '@angular/core';
import { IProduct } from '../interfaces/i-product'; // No olvides el import

@Pipe({
  name: 'productFilter'
})
export class ProductFilterPipe implements PipeTransform {
  transform(value: IProduct[], filterBy: string): IProduct[] {
    return null;
  }
}
```

A continuación, vamos a implementar el método. Vamos a comparar el parámetro **filterBy** con el nombre (description) del producto, ambos en minúsculas. Nos quedaremos con los productos que contengan la cadena del parámetro en el nombre. Si no hay cadena de entrada para filtrar, devolvemos el array original.

```
transform(products: IProduct[], filterBy: string): IProduct[] {
  const filter = filterBy ? filterBy.toLocaleLowerCase() : null;
  return filter ?
    products.filter(prod => prod.desc.toLocaleLowerCase().includes(filter)) :
    products;
}
```

Ya sólo queda aplicar el filtro. En la plantilla product-list.component.html, vamos a aplicar el filtro a los productos que recorre ***ngFor**. Añadimos el filtro productFilter, y a continuación la cadena para filtrar, en este caso definida por la propiedad filterSearch del componente (vinculada al campo de búsqueda).

```
<tr *ngFor="let product of products | productFilter:filterSearch">
  ...
</tr>
```

Y ya funciona el filtro de búsqueda en tiempo real!

Filtro:	<input type="text" value="ssd"/>	Filtrando por: ssd	
Ocultar imagen	Producto	Precio	Disponible
	SSD hard drive	€75.00	03/10/2016

Más sobre componentes

En esta sección vamos a aprender un poco más sobre los componentes. Cómo aplicar estilos sólo a un componente, y cómo funciona el ciclo de vida de un componente. En la siguiente semana veremos como anidar componentes.

Encapsulación de estilos en componentes

Como a estas alturas sabrás, al crear un componente con Angular CLI, se genera un archivo CSS asociado, que está referenciado desde el decorador `@Component`. Los estilos añadidos en dicho archivo **sólo se aplican** a este componente.

En lugar de usar un archivo CSS, se pueden poner las propiedades CSS en una cadena, usando el atributo **style** del decorador `@Component`:

```
@Component({
  selector: 'product-list',
  templateUrl: './product-list.component.html',
  styles: [
    td {
      vertical-align: middle;
    },
    td:first-child img {
      height: 40px;
    }
  ]
})
```

Sin embargo, suele ser más recomendable tener los estilos en uno o varios archivos CSS separados. Estos se referencian con la propiedad **styleUrls** (array):

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

Para aislar los estilos y que sólo se apliquen al componente actual. Angular genera unos atributos aleatorios tanto para los elementos HTML del componente como para los selectores CSS del mismo (para que sólo afecten a dichos elementos).

<pre>▼<tr _ngcontent-kfp-2> ▶<td _ngcontent-kfp-2>...</td> == \$0 <td _ngcontent-kfp-2>SSD hard drive</td> <td _ngcontent-kfp-2>€75.00</td> <td _ngcontent-kfp-2>03/10/2016</td> </tr></pre>	<pre>td[_ngcontent-kfp-2] { vertical-align: middle; }</pre>
--	---

Estilos globales

Para definir estilos globales en la aplicación, tenemos 2 opciones:

- Crearlos (o importar otros archivos) en el archivo **src/styles.css**:

```
/* You can add global styles to this file, and also import other style files */
@import "../node_modules/bootstrap/dist/css/bootstrap.css"
```

- Incluir los archivos CSS en el archivo **angular.json**:

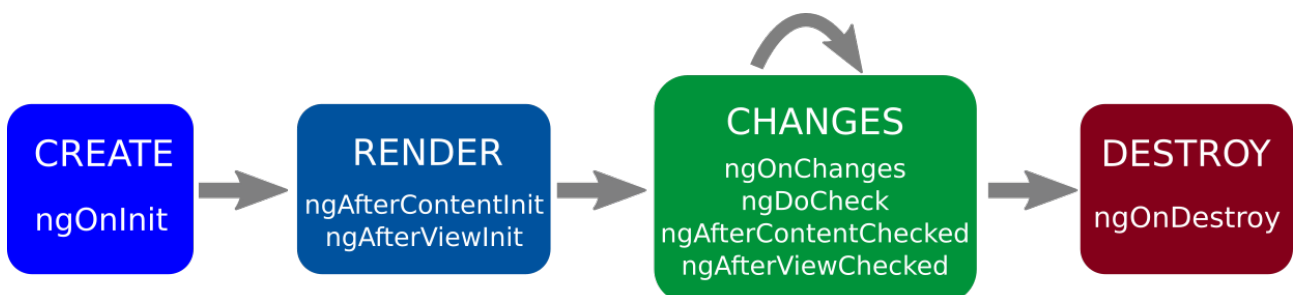
```
"styles": [
  "styles.css",
  "../node_modules/bootstrap/dist/css/bootstrap.css"
],
```

Angular CLI es capaz de trabajar directamente con preprocesadores CSS como SASS, LESS, etc. Usa la opción **--style** al crear un nuevo proyecto (valores: css, scss, less, sass, styl). Se pueden cambiar también la preferencia en el archivo **angular.json** (sección “defaults”).

```
"defaults": {
  "styleExt": "css",
  ...
}
```

Ciclo de vida de los componentes

Hay varias etapas por las que pasa un componente desde que se crea hasta que se destruye (se elimina de la memoria). Angular nos permite actuar en cada una de las fases mediante eventos.



- **ngOnInit** → Se ejecuta sólo una vez, cuando el componente se crea. Se suele utilizar para obtener datos del servidor.
- **ngAfterContentInit** y **ngAfterViewInit** → Se ejecutan una vez (en ese orden) justo después de ngOnInit. **ngAfterContentInit** se ejecuta cuando el contenido está listo para mostrarse en la vista, y **ngAfterViewInit** se ejecuta cuando el componente y todo lo que contiene se ha renderizado en el documento.
- **ngOnChanges** → Se ejecuta cada vez que cambia el valor de una propiedad vinculada a la vista (interpolación, vinculación de atributos, etc.).
- **ngDoCheck** → Este evento se puede usar para detectar cambios que Angular no detecta automáticamente. Sin embargo, este método (si se crea), se ejecuta muy a menudo y se recomienda evitar usarlo siempre que se pueda para no degradar el rendimiento de la aplicación.
- **ngAfterContentChecked** y **ngAfterViewChecked** → Se ejecutan justo

después de `ngDoCheck` en ese orden. No se suelen implementar.

- **ngOnDestroy** → Se llama justo cuando el componente se destruye (desaparece de la vista).

Más información sobre el ciclo de vida [aquí](#).

Veamos un pequeño ejemplo del evento **ngOnInit**:

```
import { Component, OnInit } from '@angular/core';
...
export class ProductListComponent implements OnInit {
  ...
  ngOnInit() {
    console.log("ProductListComponent has been initialized!");
  }
}
```

Como puedes observar, se implementa la interfaz **OnInit**, que obliga a implementar a su vez el método **ngOnInit**. Cada evento del ciclo de vida tiene su propia interfaz que podemos implementar cuando la necesitemos. La interfaz se llama igual que el método a implementar sin el prefijo '**ng**'. Por ejemplo: **ngOnDestroy** → **OnDestroy**. Estas interfaces se importan del módulo **@angular/core**.

Deberías ver un mensaje como este en la consola cuando la página (y por tanto el componente) se carga:

```
ProductListComponent has been initialized!    product-list.component.ts:40
```