

# Frontend con Angular by Pixelpro

(A medida que voy subiendo y desarrollando el curso, actualizado este documento, hasta que no desaparezca este mensaje, os recomiendo verlo online para estar al día en lugar de descargarlo.)

Repo: <https://github.com/bypixelpro/cursoangular9>

## Índice Angular Básico (1) & Intermedio (2)

- 1.0 Angular CLI, comandos
- 1.2 Arquitectura básica de una app por defecto
- 1.3 Anatomía de un Componente
- 1.4 Interacción entre componentes, plantillas, propiedades y métodos
- 1.5 Anidar Componentes
- 1.6 Enviar datos a un componente con input
- 1.7 Directivas de uso común
- 1.8 El enrutador de Angular
- 1.9 Rutas estáticas vs dinámicas
- 1.10 Servicios
- 1.11 CSS & DOM
- 1.12 Renderer 2
- 1.13 Publicando
  
- 2.1 Componentes Stateful vs Stateless
- 2.2 El maravilloso mundo de los eventos
- 2.3 Control de componentes anidados con ViewChild
- 2.4 Política detección de cambios y optimización rendimiento
- 2.5 Eventos y ciclo de vida
- 2.6 OnDestroy
- 2.7 OnChanges
- 2.8 OnDoCheck
- 2.9 Mouse events
- 2.10 Keyboard Events
- 2.11 Formularios reactivos

## 1.0 Angular CLI

Comandos habituales.

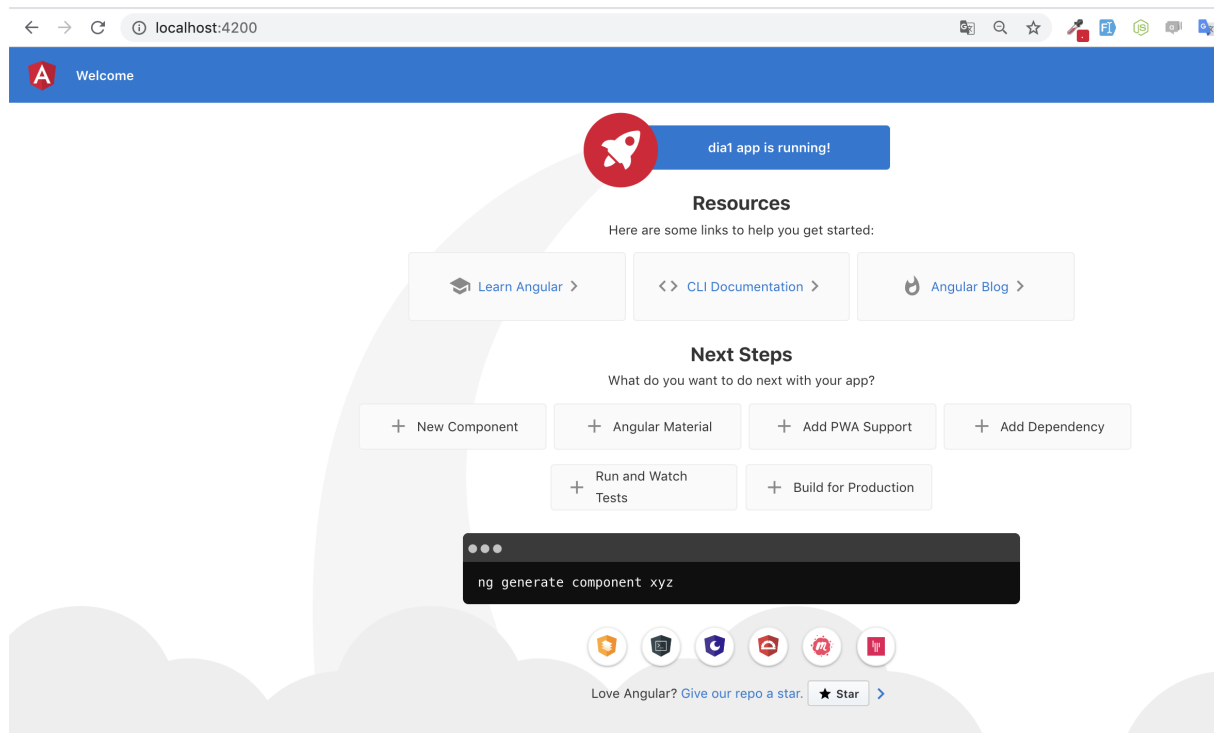
Vamos a instalar en Angular cli mediante npm.

**npm install -g @angular/cli** (Línea de comandos de angular de manera global)

**ng new** (+ nombre del proyecto + enrutador + CSS)

**ng generate component** (+ nombre componente)

**ng serve** (microservidor en node, localhost:4200 + compilado a javascript)



## 1.2 Arquitectura básica de una app por defecto:

- **e2e:** Pruebas end to end. Pruebas dentro del navegador. Examinar funcionalidades completas dentro de la app. La configuración está en: **protractor.conf.js**. El archivo: **app.e2e-spec.ts** contiene una prueba de texto, que evalúa. Para ejecutar esta prueba, usa el comando **ng e2e**.
- **node\_modules:** Todos los módulos del proyecto, podrás ver que hay un alto número. Esta carpeta no ha de ser incluida en tus proyectos. Son archivos base de desarrollo. Cuando descargues un proyecto, esta carpeta no suele estar incluida. Cuando no esté, colócate a nivel del archivo **package.json** y ejecuta: **npm install**. Esto instalará los módulos y sus dependencias.
- **src:** Aquí está nuestra app. En el index.html está el root de la app, el style.css... En la carpeta app, está el código fuente de nuestra aplicación. Aquí encontrarás tus componentes. test1 tiene la arquitectura habitual de un módulo. Encontrarás un .ts, un .html y un .css
- **assets:** esta carpeta está destinada a imágenes, archivos... incluso algún script.
- **environment:** Información sobre el entorno de trabajo que vamos a usar, por defecto producción está en false. Una vez que tengamos la app lista, pasaremos a true.
- **package.json:** Contiene la información para instalar dependencias.

- **.gitignore:** Configuración de git para ignorar archivos del sistema.

## 1.3 Anatomía de un Componente

Los componentes son en resumen bloques de construcción para nuestro proyecto. Vamos a crearnos un componente básico con: **ng generate component** (nombre random)  
Entre los archivos, encontrarás un random.component.spec.ts(test unitarios) que es para pruebas e2e. trabajaremos de forma encapsulada los estilos del componente con esta arquitectura. .ts, .html. css.

Veamos el archivo que contiene la lógica:

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

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

  constructor() { }

  ngOnInit(): void {
  }

}
```

En el import, importamos funcionalidad del core de angular. Un par de módulos en este caso (Se puede tener una cantidad indefinida de imports)

Luego tenemos los decoradores, para configurar el componente, con tres propiedades:

**selector:** invocar el componente

**templateUrl:** definimos el url de la plantilla

**styleUrls:** sección de hojas de estilo del componente

Pasamos luego a definir la clase, donde definimos el componente. **Constructor** y **ngOnInit**.

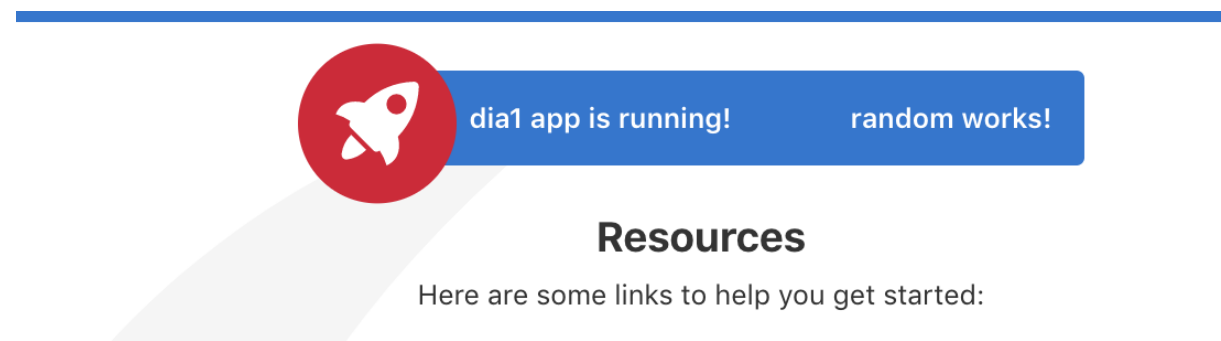
La diferencia entre estos dos eventos es que uno se ejecuta al inicio de la aplicación y otro en el momento en que comenzamos a utilizar nuestro componente.

Aquí podemos incluir una cantidad indefinida de métodos para poder trabajar nuestra aplicación y también podemos incluir diferentes variables.

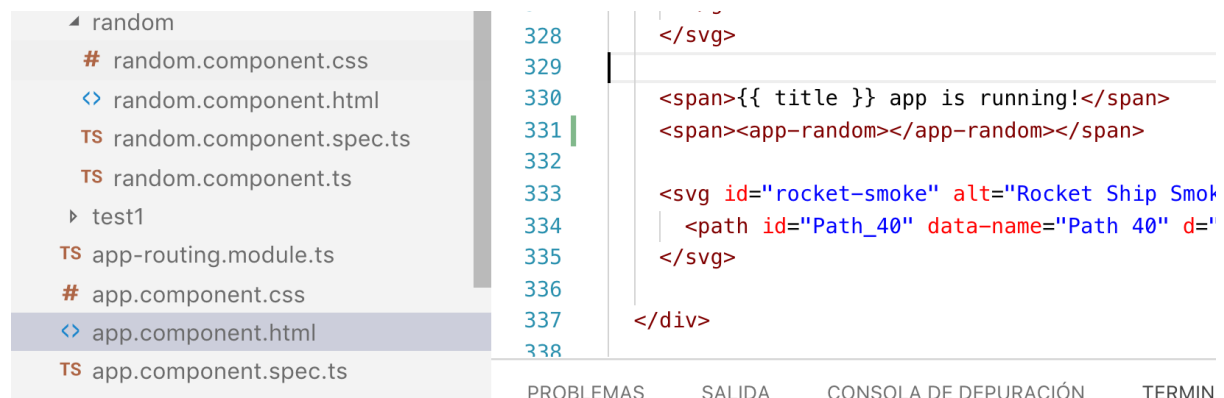
Una vez añadimos un módulo, este no se carga por defecto. examinemos el index.html y el app-root.

**Ejercicio:** Muestra debajo del saludo de la app por defecto, el texto de tu componente:

```
<p>random works!</p>
```



**Solución:**



Fácil ¿no? Ahora muestra un número aleatorio desde tu componente para introducir algo de dinamismo.

## 1.4 Interacción entre componentes, plantillas, propiedades y métodos

Un componente en su .ts habla con la plantilla de la siguiente forma:

```
TS random.component.ts
1 import { Component, OnInit } from '@angular/core';
2
3 @Component({
4   selector: 'app-random',
5   templateUrl: './random.component.html',
6   styleUrls: ['./random.component.css']
7 })
8 export class RandomComponent implements OnInit {
9
10   numrandom: Number = Math.floor(Math.random()* 10);
11
12   constructor() { }
13
14   ngOnInit(): void {
15   }
16
17 }
18
```

```
<> random.component.html dia1/src/app/random
1 <p>{{numrandom}}</p>
2
```

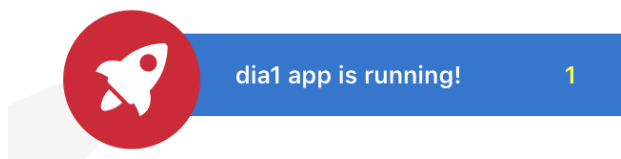
En angular usamos {{}} para mostrar valores. Por ejemplo: **nombre:string = 'David'**;

También podemos complicarlo un poco y usarlo para mostrarlo con una directiva.

```
TS random.component.ts
1 import { Component, OnInit } from '@angular/core';
2
3 @Component({
4   selector: 'app-random',
5   templateUrl: './random.component.html',
6   styleUrls: ['./random.component.css']
7 })
8 export class RandomComponent implements OnInit {
9
10   numrandom: Number = Math.floor(Math.random()* 10);
11   miamarillo:String = 'Yellow';
12
13   constructor() { }
14
15   ngOnInit(): void {
16   }
17
18 }
19
```

```
<> random.component.html dia1/src/app/random
1 <p [ngStyle]="{'color': miamarillo}">{{numrandom}}</p>
2
```

El resultado será:



Vamos a probar ahora con algo más elaborado. Nos creamos un método que genere un color aleatorio y le pasamos el método como valor de la propiedad::

```
TS random.component.ts
6 | styleUrls: ['./random.component.css']
7 | })
8 | export class RandomComponent implements OnInit {
9 |
10 |   numrandom: Number = Math.floor(Math.random()* 10);
11 |   miamarillo:String = 'Yellow';
12 |
13 |   constructor() { }
14 |
15 |   ngOnInit(): void {
16 |   }
17 |
18 |   generarRandom():String{
19 |     return Math.floor(Math.random()*255).toString(16)
20 |   }
21 |
22 |   colorHex():String{
23 |     return '#' + this.generarRandom()+this.generarRandom()+this.generarRandom()
24 |   }
25 |
26 | }
27 |

random.component.html dia1/src/app/random
1 | <p [ngStyle]='{"color": colorHex()}'>{{numrandom}}</p>
2 |
```

## 1.5 Anidar Componentes

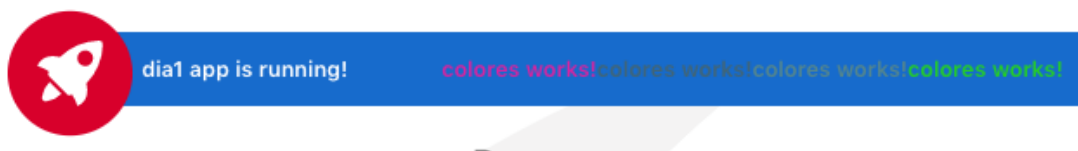
Vamos a hacer que el componente por defecto, incluya a otros componentes y conseguir así una planteamiento más modular.

**ng generate component colores**

Ahora, pasa el generador de colores, a este componente y haz que se muestre en otro sitio del componente principal. Si duplico la vista, el resultado se ejecuta cada vez. Cada componente trabaja de manera encapsulada e independiente.

```
<span>{{ title }} app is running!</span>
<span></span>

<app-colores></app-colores>
<app-colores></app-colores>
<app-colores></app-colores>
<app-colores></app-colores>
```

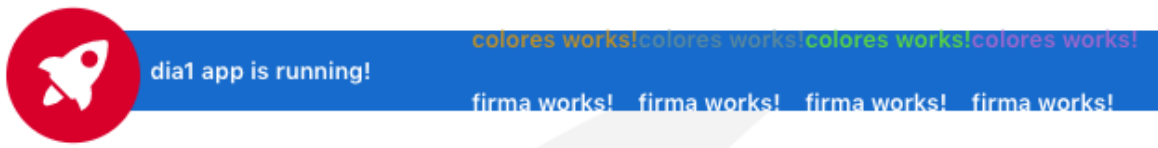


Para seguir rizando el rizo, vamos a generar otro componente que se llame firma:  
**ng generate component firma**

Y añadiremos la vista de este a la de colores:

```
<> colores.component.html dia1/src/app/colores
1 | <p [ngStyle]='{"color": colorHex()}'>colores works!</p>
2 | <app-firma></app-firma>
3 |
```

El resultado será la anidación de un componente, dentro de otro anidado:





## 1.6 Enviar datos a un componente con input

Hasta el momento, cada componente trabaja de manera encapsulada, ahora vamos a enviar información desde el componente principal hasta el anidado:

- 1) Modifica la vista de la firma con un texto estático: `<p>xxx</p>`
- 2) Vamos a **colores.component.ts** una vez aqui, convierte el color hexadecimal que se genera ahora en una variable local
- 3)

```
export class ColoresComponent implements OnInit {  
  
  colorLocal: string;  
  
  constructor() { }  
  
  ngOnInit(): void {  
  }  
  
  generarRandom(): String {  
    return Math.floor(Math.random() * 16777215).toString(16);  
  }  
  
  colorHex(): String {  
    this.colorLocal = '#' + this.generarRandom();  
    return this.colorLocal;  
  }  
}
```

- 4) Vamos a la plantilla. Nos preparamos la vista para recibir un parámetro

```
<> colores.component.html dia1/src/app/colores  
```

```
1 <p [ngStyle]="{'background': colorHex()}">
2 <app-firma [colorNombre] = "colorLocal"></app-firma>
3 </p>
```

- 5) En el componente firma, importamos un Input para poder trabajar con este. Para invocarlo, usamos **@Input() colorNombre:string**

TS firma.component.ts primerospasos/src/app/firma

```
1 import { Component, OnInit, Input } from '@angular/core';
2
3 @Component({
4   selector: 'app-firma',
5   templateUrl: './firma.component.html',
6   styleUrls: ['./firma.component.css']
7 })
8 export class FirmaComponent implements OnInit {
9
10   @Input() colorNombre: string;
11
12   constructor() { }
13
14   ngOnInit(): void {
15   }
16
17 }
18
```

- 6) Finalmente, vamos a modificar la vista, introduciendo la variable.

```
<> firma.component.html dia1/src/app/firma
```

```
1 <p>{{colorNombre}}</p>
2 |
```



dia1 app is running!

#2ab17#b136a#5d2079#f1a2a5

- 7)

Resources



## 1.7 Directivas de uso común

**NGIF:** Esta directiva nos permite mostrar o desplegar elementos. En esta versión frente a angularjs se mejora considerablemente el rendimiento, ya que no carga de verdad.

veamos un ejemplo:

Creamos una variable en el componente principal de la aplicación: `condicion:Boolean;`

Y luego añadimos el ngIf en la vista junto con un escuchador del evento click, para mostrarlo

```
<div class="content" role="main">
  <button (click)="condicion = true">Ver</button>
  <div *ngIf="condicion">
    <!-- Highlight Card -->
    <div class="card highlight-card card-small">
```

Con esto conseguimos un botón que muestra o no el elemento. Importante, vamos a inspeccionar para ver que hace a nivel código (hacemos un debug)

Seguramente estarás pensando cómo hacer que aparezca y desaparezca. Vamos a usar un viejo truco:

```
<button (click)="condicion = !condicion">
```

Listo! Con esto conseguimos exactamente lo que quieres, mostrar y ocultar.

Ahora tu, muestra el valor de la variable en la vista:

Ver  
Valor: false

### Resources

Here are some links to help you get started:

**NGELSE:** Importante saber, que cuando añadimos una directiva como ngIf, esta solo tiene alcance dentro de la etiqueta en la que se encuentra `<div>`.

Si queremos añadir la lógica de else, lo haremos de la siguiente forma:

```
<div *ngIf="condicion ; else textoalternativo">
```

Ese texto alternativo, en versiones anteriores (ang 4) se mostraba dentro de la etiqueta `template`. Esta representaba una incompatibilidad la la etiqueta de html5 En esta versión usamos `<ng-template>`.

```
<ng-template #textoalternativo>Texto Alternativo </ng-template>
```

Este recurso, lo usamos por ejemplo en instagram para mostrar contenido explicito que ha de ser aceptado previamente. Para tareas más complejas, usaremos otros recursos como el `switch`.

## NGSWITCH:

```
<h2>>Ejemplo NGSWITCH</h2>
<div>
<p>Texto de ejemplo</p>
</div>
<input type="text">
```

Comenzamos con un poco de html y seguidamente después vamos a añadirle la directiva en la etiqueta correspondiente y le asignamos el valor que queremos evaluar, en este caso nombre:

```
<div [ngSwitch]="Nombre">
```

Nombre evaluará una serie de datos, que vamos a introducir dentro de ngModel, un modelo en angular no es más que una variable, en este caso esa variable esta relacionada a lo que escriba el usuario.

```
<input type="text" [(ngModel)]="Nombre">
```

Las siguientes directivas que un switch necesita para operar son:

```
[ngSwitch]="so"
*ngSwitchCase="'Windows'"
*ngSwitchDefault
[(ngModel)]="so"
```

Vamos a ver un ejemplo práctico, un sencillo programa que nos permita mostrar información en base a lo que introduzca el usuario.

### En app.component.ts

```
<ul [ngSwitch]="so">
  <li *ngSwitchCase="'Windows'">El que usan todos</li>
  <li *ngSwitchCase="'Mac'">Los que les gusta lo bonito, tienen
pasta</li>
  <li *ngSwitchCase="'Linux'">Programador de siempre</li>
  <li *ngSwitchDefault>Indica un S.O.</li>
</ul>

<input type="text" [(ngModel)]="so"/>
```

Luego en esta versión hay que importar en el modelo lo siguiente:

En **app.module.ts**

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

imports: [
  BrowserModule,
  AppRoutingModule,
  FormsModule
],
```

**NGSTYLE & NGCLASS:** Dos directivas muy útiles que se usan a menudo. ngStyle es una directiva que nos permite inyectar directamente estilos en un elemento html.

```
<div [ngStyle]="{'background': 'red'}">Texto Fondo Rojo</div>
```

También podemos añadir algo de lógica a esta directiva:

```
<div [ngStyle]="{'background':color ? color:'red'}">Texto Fondo Rojo</div>
```

Aquí vemos una sintaxis similar a la de un operador ternario, donde tenemos una lógica sencilla, usa de fondo color, y si este no está definido usa color:red. Luego añadimos botones para modificar su valor:

```
<div [ngStyle]="{'background':color ? color:'red'}">Texto Fondo Rojo</div>

<button (click)="color = 'yellow'">Amarillo</button>
<button (click)="color = 'blue'">Azul</button>
<button (click)="color = 'purple'">Morado</button>
```

Como puedes intuir, con ngClass, en lugar de insertar las propiedades directamente, lo que hacemos es asignar una clase de css:

```
<div [ngStyle]="{'background':color ? color:'red'}">Texto Fondo Rojo</div>

<button (click)="color = 'yellow'"
[ngClass]="color">Amarillo</button>
<button (click)="color = 'blue'"
[ngClass]="color">Azul</button>
```

```
<button (click)="color = 'purple'"
[ngClass]="color">Morado</button>
```

Ahora cada vez que presiones un botón, todos tendrá la clase seleccionada. escribe en tu hoja de estilos esa clase y mira el resultado.

Podemos darle una vuelta más. Haz que solo se muestre en color el boton que lleva por nombre el color activo.

```
<button (click)="color = 'yellow'" [ngClass]="{'activoyellow':
color == 'yellow'}">Amarillo</button>
```

Con esto le decimos, en el click, añade fondo amarillo, y además si el fondo es amarillo, añade la clase **activoyellow**.

**NGFOR:** Es muy sencillo de implementar, básicamente lo que hace es iterar sobre una colección e instanciarlo. Veamos un ejemplo sencillo:

En el componente, creamos un array que va a contener objetos.

```
libros:Array<object>;
```

Y seguidamente después en el constructor lo implementamos:

```
constructor() {
  this.libros = ["Harry potter", "Los 7 habitos", "La celestina"]
}
```

Ahora solo tenemos que iterar la colección en la vista:

```
<ul>
  <li *ngFor="let libro of libros">{{libro}}</li>
</ul>
```

Libro será la instancia de cada uno de los elementos del array. Al definir este array como contenedor de objetos, podemos hacer uso de las propiedades de los objetos para componer listas más complejas:

```
constructor() {
  this.libros = [
    {id:'1', titulo: 'Te veré bajo el hielo', autor:'Robert
Bryndza'},
    {id:'2', titulo: 'Dime quién soy', autor:'Julia Navarro'},
```

```

    {id:'3', titulo: 'El día que se perdió la cordura',
autor:'Javier Castillo'}
  ]
}

```

Ahora nuestra aplicación muestra todo el objeto, cada uno de los que estamos llamando en cada iteración. Seguramente este no será el resultado que estamos buscando así que modificaremos la vista con:

```

<ul>

  <li *ngFor="let libro of libros">{{libro.titulo}}</li>

</ul>

```

Ahora tu, muestra la lista, y que cada título esté acompañado de su autor.

Veamos a continuación cómo incorporar algunos operadores adicionales al ngFor. Uno muy recurrido es un contador de javascript para los elementos de las lista, esto lo conseguimos con:

```

<li *ngFor="let libro of libros; index as
indice">{{indice}}-{{libro.titulo}}</li>

```

Dado que JS empieza a contar desde 0, el primer título tendrá ese valor.

Para ver si estás asimilando bien los conceptos, vamos a combinar ngFor con ngIf.

Ejercicio: En el elemento con el valor 1 de índice, muestra una etiqueta que diga “Best Seller”.

```

<ul>

  <li *ngFor="let libro of libros; index as indice">
    {{indice}}-{{libro.titulo}}<span *ngIf="indice ==1"> - Best
Seller"</span>
  </li>

</ul>

```

También podemos hacer uso de otros operadores no ordinarios, por ejemplo first, que devuelve true o false y como ya te estarás imaginando, selecciona al primer elemento del array.

```

<ul>

  <li *ngFor="let libro of libros; index as indice ; first as
primero">
    {{indice}}-{{libro.titulo}}
    <span *ngIf="primero"> - Best Seller"</span>

```

```
    </li>
</ul>
```

De la misma manera existe el operador **last** (último elemento) **odd** (es par) **even** (es impar). Pruébalo.

Lo siguiente que veremos es como añadir enlaces y eventos a nuestras lista. Un truco sencillo para que cada elemento de lista nos lleve a la búsqueda en google del libro sería:

```
<ul>
  <li *ngFor="let libro of libros">
    <a
href="https://www.google.com/search?q={{libro.titulo}}">{{libro.ti
tulo}}</a>
  </li>
</ul>
```

Algo más dinámico, son los eventos. Ahora cuando se hace click en un libro nos indica su autor.

En la vista vamos a pasar el elemento pinchado al componente:

```
<ul>
  <li *ngFor="let libro of libros">
    <a (click)="showAuthor(libro)">{{libro.titulo}}</a>
  </li>
</ul>
```

Y en el componente:

```
showAuthor(_libro){
  this.verAutor = 'Escrito por: ' + _libro.autor;
  alert(this.verAutor)
  console.log(_libro.titulo, 'escrito por', _libro.autor)
}
}
```

## 1.8 El enrutador de Angular

Una de las características más populares que tiene Angular es su capacidad de generar aplicaciones dentro de un solo documento. Por regla general, cuando trabajamos una aplicación Angular, aunque nosotros tenemos un buen número de vistas, componentes..., al final cuando se despliega la aplicación en el navegador del usuario lo que va a tener es básicamente un solo archivo. Dentro de ese archivo se van a ir cargando todos los demás componentes de manera asíncrona. Una vez que nosotros tenemos una página o una aplicación utilizando un solo documento, a pesar de que tenemos una eficiencia mejorada vamos a tener un problema con la indexación.

Angular nos va a permitir a nosotros generar también indexación a través de rutas y, por ejemplo, a través de diferentes rutas acceder a los contenidos, a pesar de que se encuentran dentro de un solo documento, mostrar diferentes partes de este documento y a la vez indexarlos dentro de los diferentes motores de búsqueda, lo cual significa que nuestra aplicación no solo va a ser muy eficiente, sino que también se va a indexar correctamente en los diferentes navegadores. Ahora vamos a aprender cómo hacer esto.

Crea un componente que se llame “sobre nosotros” otro llamado “libros” y finalmente “cabecera”.

Luego vamos al módulo principal de la app: `app.module.ts` allí deberemos importar el enrutador si no está y establecer la rutas en **`app.module.ts`**:

```
imports: [
  BrowserModule,
  AppRoutingModule,
  FormsModule,
  RouterModule.forRoot(rutas)
],
.
import { RouterModule, Routes } from '@angular/router';
```

Luego nos creamos una constante para almacenar las rutas:

```
const rutas:Routes = [
  {path:'listado-libros', component: LibrosComponent}
]
```

Y lo importamos junto con el método `forRoot()`:

```
imports: [
```

```
BrowserModule,  
AppRoutingModule,  
FormsModule,  
RouterModule.forRoot(rutas)  
],
```

En la vista, **app.component.html** insertamos lo siguiente:

```
<app-cabecera></app-cabecera>  
<router-outlet></router-outlet>
```

Con esto, conseguimos tener siempre una cabecera en toda la vida de la ejecución de nuestra app, y el contenido dinámico cargado por el enrutador.

Ahora vamos a ver cómo definir rutas por defecto.

```
{path:'', component: InicioComponent, pathMatch: 'full'}
```

Crea un componente y en las rutas, agrega la propiedad pathMatch full, con esto definimos donde tiene que ir la app por defecto.

Nos falta manejar los 404, es decir los errores de url cuando un usuario se equivoca.

```
{path:'**', redirectTo: '/'}
```

Ejercicio: Crea un componente y una ruta que se encargue de mostrar un mensaje cuando se produce un error de página no encontrada.

## 1.9 Rutas estáticas vs dinámicas

Podemos crear una navegación sencilla en el componente de cabecera:

```
<ul>  
  <li><a href="/listado-libros">Listado libros</a></li>  
  <li><a href="/">Inicio</a></li>  
</ul>
```



El problema con este planteamiento, es que estamos desperdiciando toda el potencia de angular:

The screenshot shows a web browser at `localhost:4200/listado-libros`. The application has a blue header with a 'Welcome' message and a Twitter icon. Below the header, there's a form with a 'Ver' button, a 'Valor:' label, a 'Texto Alternativo' label, and a list of items: 'Indica un S.O.', '0-Te veré bajo el hielo - Best Seller\*', '1-Dime quién soy - Best Seller\*', and '2-El día que se perdió la cordura'. There are also color selection buttons (Amarillo, Azul, Morado) and a 'hola works!' message. The network tab on the right shows a list of requests, including 'websocket', 'listado-libros', 'runtime.js', 'polyfills.js', 'styles.js', 'vendor.js', 'main.js', 'data:image/svg+xml...', 'info?t=1583347456586', and 'inject.js'. The 'listado-libros' request is highlighted, showing a 304 status and a size of 210 B.

Si inspeccionamos, veremos que al hacer click en cada elemento del menú, volvemos a cargar toda la app, elementos repetidos...

Veamos cómo usar las capacidades asíncronas de angular para evitar toda esa carga de datos innecesaria.

```
<ul>
  <li><a [routerLink]="['/listado-libros']" >Listado
libros</a></li>
  <li><a href="/">Inicio</a></li>
</ul>
```

Prueba a limpiar el network, y volver a cargar la app, una vez estés en el inicio, haz click en listado libros y verás que no hay carga, aunque el resultado es el mismo que antes.

Si quieres añadir una clase al elemento activo usa:

```
<ul>
  <li><a [routerLink]="['/listado-libros']"
routerLinkActive="myclass">Listado libros</a></li>
  <li><a href="/">Inicio</a></li>
```

```
</ul>
```

Si inspeccionas el elemento, verás que tiene esa clase añadida.

## Vamos con los enlaces dinámicos

Esto suele ser lo más habitual. Vamos a preparar el componente de destino:

Podemos hacer algo así en la vista del componente libro:

```
<ul>
  <li *ngFor="let libro of libros">
    <a [routerLink]="['/informacion']">{{libro.titulo}}</a>
  </li>
</ul>
```

Para que esta vista funcione, el componente de libros, debe tener este aspecto:

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

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

  libros:Array<object>;

  constructor() {
    this.libros = [
      {id:'1', titulo: 'Te veré bajo el hielo', autor:'Robert
Bryndza'},
      {id:'2', titulo: 'Dime quién soy', autor:'Julia Navarro'},
      {id:'3', titulo: 'El día que se perdió la cordura',
autor:'Javier Castillo'}
    ]
  }

  ngOnInit(): void {
```

```
}  
}
```

Ahora, cada enlace, nos llevará a /información, lo cual no es una buena práctica, ya que no puedo indexar esas páginas, no podría compartir esa url ni saber donde esta o que está viendo el usuario de forma sencilla.

Esto lo solucionaremos enseguida, de momento vamos a modificar el enrutador:

```
{path: 'informacion/:libroId', component: InformacionComponent},
```

Agregamos un url con una variable y destinamos a un nuevo componente.

```
{path: 'informacion/:libroId', component: InformacionComponent},  
{path: 'informacion', redirectTo: '/'},
```

(Nota que la genérica va por debajo, de lo contrario se interpone)

No dejamos la url de información sin manejar, podemos redirigirla a donde queramos.

Finalmente y para que todo tenga sentido, vamos a enviar el id del libro del componente al enrutador. Vamos a la vista de libros:

```
<h2>Listado Libros:</h2>  
<ul>  
  <li *ngFor="let libro of libros">  
    <a [routerLink]="['/informacion',  
libro.id]">{{libro.titulo}}</a>  
  </li>  
</ul>
```

Como el valor de router link es un array, puedo enviar más cosas. En este caso un parámetro.

Finalmente, hay que mostrar el contenido de cada libro. El objetivo es que el componente de destino, sepa qué información ha de mostrar en base al parámetro de la ruta.

Como de momento no sabemos usar los servicios para pasar info de una componente a otro, vamos a replicar el array con la info de los libros, dentro de la página de destino, de modo que leeremos la url y mostraremos el valor que coincida (id)

En **informacion.component.ts**

```
libros: Array<object>;  
constructor() {  
  this.libros = [  
    {id: '1', titulo: 'Te veré bajo el hielo', autor: 'Robert  
Bryndza'},  
    {id: '2', titulo: 'Dime quién soy', autor: 'Julia Navarro'},
```

```

        {id:'3', titulo: 'El día que se perdió la cordura',
autor:'Javier Castillo'}
    ]
}

```

Ahora importamos:

```
import {ActivatedRoute, Params} from '@angular/router';
```

Con el primero, conoceremos la ruta en la que se encuentra el usuario, con el segundo, los parámetros que hay en esa ruta.

En el constructor, añadiremos una variable privada con la ruta del usuario:

```
constructor( private rutausuario:ActivatedRoute){
```

Luego en el evento onInit, de manera asíncrona, ya que cuando el componente se carga, es posible que aún no tengamos el valor del parámetro esperado y nos quedemos en blanco, lo que hacemos es suscribirnos a un observable y cuando lo tengamos, entonces actuamos.

Vamos a usar un poco de JS o TS para encontrar ese libro id

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

```

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

  libros:Array<object>;
  libroId:number;
  libroClick:object;

  constructor( private rutausuario:ActivatedRoute){

    this.libros = [
      {id:'1', titulo: 'Te veré bajo el hielo', autor:'Robert
Bryndza'},
      {id:'2', titulo: 'Dime quién soy', autor:'Julia Navarro'},

```

```

        {id:'3', titulo: 'El día que se perdió la cordura',
autor:'Javier Castillo'}
    ]
}

ngOnInit(): void {
    this.rutausuario.params.subscribe(params =>{ //Params aquí, es
un método asíncrono, que tiene subscribe
        this.libroId= params['libroId'] //Params aqui es un array
asociativo
        this.libroClick = this.libroBuscador();
    })
}
filtroId(libro){
    return libro.id == this;
}

libroBuscador(){
    return this.libros.filter (this.filtroId, this.libroId) [0];
}

}

```

Ahora en la vista, mostramos el resultado:

```

<h2>Libro: {{libroClick.titulo}}</h2>
<span>Escrito por: {{libroClick.autor}}</span>

```

Como ves esto funciona pero no es la mejor forma, vamos a hablar ahora acerca de los servicios.

## 1.10 Servicios

Un servicio, en nuestro caso, nos va a permitir, saber por dónde ha ido pasando el usuario sin caer en el típico planteamiento de usar una variable global o el localhost, que sería contrario al planteamiento modular o encapsulado que tenemos.

**ng generate service libroclicked** luego vamos al módulo e introducimos:

```
import { LibroclickedService } from './libroclicked.service';

providers: [
  LibroclickedService
],
```

Ya tenemos el servicio implementado en el módulo, ahora vamos a definir la lógica de este:

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

@Injectable({
  providedIn: 'root'
})
export class LibroclickedService {

  libros: Array<object>

  constructor() {
    this.libros = [];
  }

  libroVisto(libroVisto){
    this.libros.push(libroVisto);
  }

  verListado(){
    if (this.libros.length > 0){
      return this.libros;
    }else{
      return false;
    }
  }
}
```

Una vez que hemos definido la lógica, lo siguiente es implementar nuestro servicio, dentro del componente que lo deba usar. Para eso, vamos a ampliar un poco **libros.componente.ts**

Con lo siguiente, pretendemos cargar el listado de libros, desde un recurso externo. En angular y en general en JS trabajamos mucho con el formato JSON, ya que es muy ligero.

Vamos a crear un método llamado `cargarLista()` y aquí, hacemos una petición http a una url. En nuestro caso es un archivo dentro de mis archivos pero se puede hacer lo mismo con una URL.

Importante, tras `http.get` usamos `subscribe`, con esto lo que hacemos es esperar hasta que el archivo nos llegue y a partir de ese punto trabajar con los datos. Esto es un observable.

```
import { Component, OnInit } from '@angular/core';
import { HttpClient, HttpResponse } from '@angular/common/http';

import { LibroclickedService } from '../libroclicked.service';

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

  libros;
  errorHttp: Boolean;

  constructor(private http: HttpClient , public
Libroclicked:LibroclickedService){
  }

  ngOnInit(): void {

    this.cargarLista();
  }

  cargarLista() {
```

```

    this.http.get('assets/lista-libros.json').subscribe(
        //Lo que este dentro de las llaves, es lo que se va a
        ejecutar una vez tengamos una respuesta de libros. La primera si
        todo esta ok, la segunda, si hay un error

        (respuesta: Response) => { this.libros = respuesta;},
        //(respuesta: Response) => { console.log('Error: ', respuesta )
        (respuesta: Response) => { this.errorHttp = true }
    )
}

agregarLibro(_libroVisto){
    this.LibroClicked.libroVisto(_libroVisto);
}
}

```

Como puedes observar, lo primero que hemos hecho es cargar el módulo HttpClient & HttpResponse. Con esto hacemos un get a un archivo en formato JSON donde tendremos la lista de libros, ids, autores...

**Ejercicio una vez termines y todo funciona ok:** En este tipo de planteamientos es importante tener un manejador de errores http, ya que si no lo implementamos, y llamamos de forma errónea al archivo, o el servidor no responde... se hace la petición, la lista se muestra vacía y nadie se entera de lo que ha pasado. Prueba a dejar como buena, la segunda línea, la que está comentada y provoca un error en la petición. El ejercicio consiste en que si el error devuelto es un 404, muestres al usuario un mensaje en pantalla.

Para poder usar esta nueva versión de módulo (obsoleto el Http desde la 4.5) Tenemos que importar en el módulo principal de nuestra app lo siguiente **app.module.ts** (No se te olvide importarlo abajo también):

```
import { HttpClientModule } from '@angular/common/http';
```

El JSON lo tenemos dentro de assets: lista-libros.json

<https://github.com/bypixelpro/cursoangular9/blob/master/lista-libros.json>

Con esto salimos por completo casi del elemento array duplicado, ahora lo hacemos mucho mejor, nos suscribimos de forma asíncrona a una ruta. Desde Angular 6, todas las respuestas de este módulo se devuelve en JSON.



Seguimos, ahora vamos a hacer que la vista, cada vez que el usuario hace click en un libro, este click quede registrado.

```
<h2>Listado Libros:</h2>
<ul>
  <li *ngFor="let libro of libros">
    <a (click)="agregarLibro(libro)"
[routerLink]="['/informacion', libro.id]">{{libro.titulo}}</a>

    </li>
  </ul>
```

Finalmente en la cabecera, vamos a ir mostrando los libros que se han clickado:

```
import { Component, OnInit } from '@angular/core';
import { LibroclickedService } from '../libroclicked.service';

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

  constructor(public librosVistos:LibroclickedService) { }

  ngOnInit(): void {
  }

}
```

Y mostramos la lista en la vista:

```
<ul>
  <li><a [routerLink]="['/listado-libros']"
routerLinkActive="myclass">Listado libros</a></li>
```

```

    <li><a href="/">Inicio</a></li>
</ul>

<h2>Libros vistos</h2>
<ul *ngIf="librosVistos.verListado()">
    <li *ngFor="let libro of
librosVistos.verListado()">{{libro.titulo}}</li>
</ul>

```

Con esto, tenemos dos componentes hablando entre si, mediante un servicio sin romper el encapsulamiento de Angular.

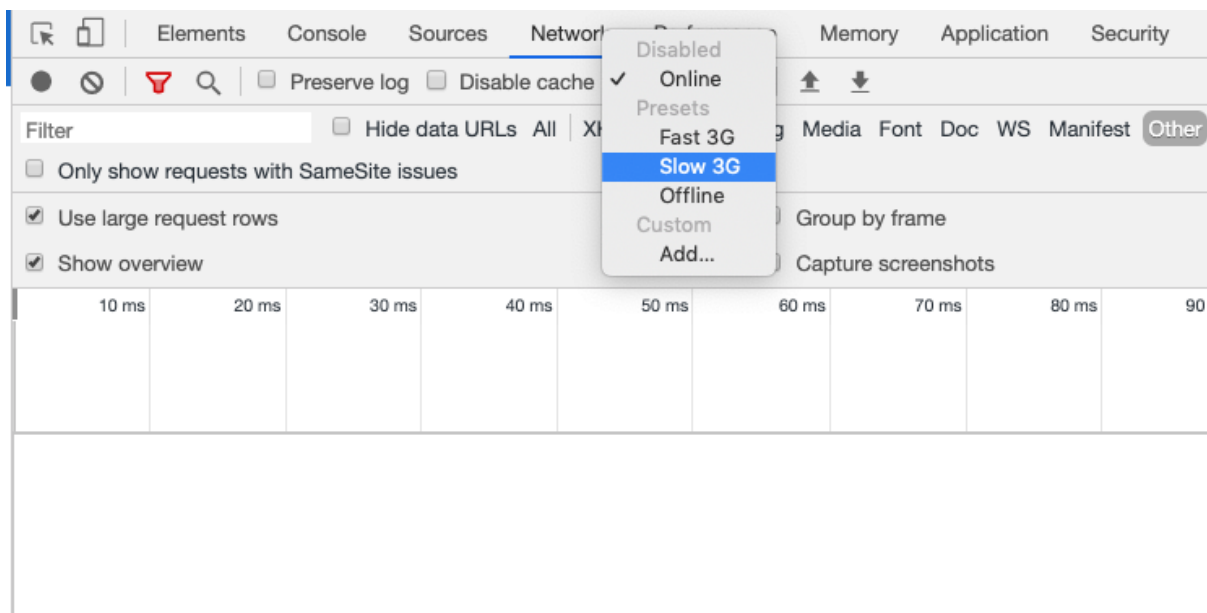
Haz ahora el ejercicio de si hay un error, mostrar un mensaje.

```

<div *ngIf="errorHttp">Hay un error en la carga de libros</div>

```

**Ejercicio 2:** Implementa un cargador, para que mientras llega la info, se muestre. Como JSON carga muy muy rápido, vamos a configurar Chrome de la siguiente forma:



Puedes encontrar un cargador en: [https://www.w3schools.com/howto/howto\\_css\\_loader.asp](https://www.w3schools.com/howto/howto_css_loader.asp)

## 1.11 CSS & DOM

Vamos a hablar ahora un poco acerca del DOM y las hojas de estilos. Cuando hablamos de encapsulación, hablamos de independencia. Prueba una cosa, crea dos componentes con un elemento h1, añade una regla para esta en el primer componente y mira como se comporta.

Inspecciona en chrome tanto la parte HTML como la CSS para entender lo que está sucediendo.

Ahora, si queremos estilos de manera global, usaremos: **src/styles.css**

Si queremos añadir un framework base tipo bootstrap, deberemos instalarlo mediante comandos. Para ello usaremos npm una vez más, así que simplemente ejecutaremos lo siguiente

**npm install bootstrap**

Luego tendremos que modificar el fichero angular.json, para extender la sección styles que quedará como sigue a continuación.

```
“styles”: [  
  “node_modules/bootstrap/dist/css/bootstrap.css”,  
  “src/styles.css”  
],
```

## 1.12 Renderer 2

Cuando CSS se nos queda corto, tenemos la clase Renderer2 que nos ayuda con un buen número de métodos adicionales.

Veamos un ejemplo. En un componente, crea una lista.

```
import { Component, OnInit, Renderer2 } from '@angular/core';
```

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

```
export class RendererdemoComponent implements OnInit {
```

```
  alumnos:Array<object>
```

```
  constructor(private ren:Renderer2) {  
    this.alumnos = [  
      {nombre: "David", id: 1},  
      {nombre: "Daniel", id: 2},  
      {nombre: "Jose", id: 3},
```

```

        {nombre: "Tamara", id: 4}
    ]
}

ngOnInit(): void {
}

}

```

Vamos a importar el `Renderer2` y en el constructor, lo cargamos dentro de una variable privada. Ahora, ya tenemos acceso a sus métodos.

Lo siguiente es la vista:

```

<ul>
    <li *ngFor="let alumno of alumnos">{{alumno.nombre}}</li>
</ul>

```

Vamos a pasar un método al evento click:

```

<ul>
    <li *ngFor="let alumno of alumnos" {{alumno.nombre}}
    (click)="activeMethod()">{{alumno.nombre}}</li>
</ul>

```

Para poder modificar el HTML, necesitamos acceso al DOM, para poder resolver esto tenemos una opción muy sencilla, añadir un identificador (una variable en realidad) con **#loquesea**. Esto convertirá la instancia del elemento en una variable local. Para usarla la inyectamos en el método que estamos usando:

```

<ul>
    <li *ngFor="let alumno of alumnos"
    (click)="activeMethod(elementoDOM)"
    #elementoDOM>{{alumno.nombre}}</li>
</ul>

```

Ahora creamos el método en el componente:

```

activeMethod(elementoDOM: HTMLElement) {

```

```
        this.ren.addClass(element, 'miclase')

    }
}
```

Y definimos la clase en el CSS:

```
li{
    transition: 0.3s all;
}

.miclase{
    background: #00D9FF;
    padding: 10px;
    color: white;
}
```

Ahora, de la misma forma que podemos agregar una clase, podemos hacer cualquier cosa, añadir un nodo, nuevos atributos y un sin fin de posibilidades... Importante, tu puedes hacer un **document.getelementbyid** y te funcionará exactamente igual, pero esto rompería el encapsulamiento de angular.

Vamos a ver cómo extender un poco más el funcionamiento de lo que tenemos.

Supongamos que quiero ahora, que al hacer click, solo el elemento clickado tenga esa clase destacada, podría hacer algo así:

```
import { Component, OnInit, Renderer2 } from '@angular/core';

@Component({
    selector: 'app-rendererdemo',
    templateUrl: './rendererdemo.component.html',
    styleUrls: ['./rendererdemo.component.css']
})
export class RendererdemoComponent implements OnInit {
    alumnos:Array<object>
    clearElement:HTMLElement

    constructor(private ren:Renderer2) {
        this.alumnos = [
            {nombre: "David", id: 1},
```

```

        {nombre: "Daniel", id: 2},
        {nombre: "Jose", id: 3},
        {nombre: "Tamara", id: 4}
    ]
}
ngOnInit(): void {
}
activeMethod(element:HTMLInputElement){

    if (this.clearElement){
        this.ren.removeClass(this.clearElement, 'miclase')
    }
    console.log('Elemento clickado!')
    this.ren.addClass(element, 'miclase')
    this.clearElement = element;
}
}

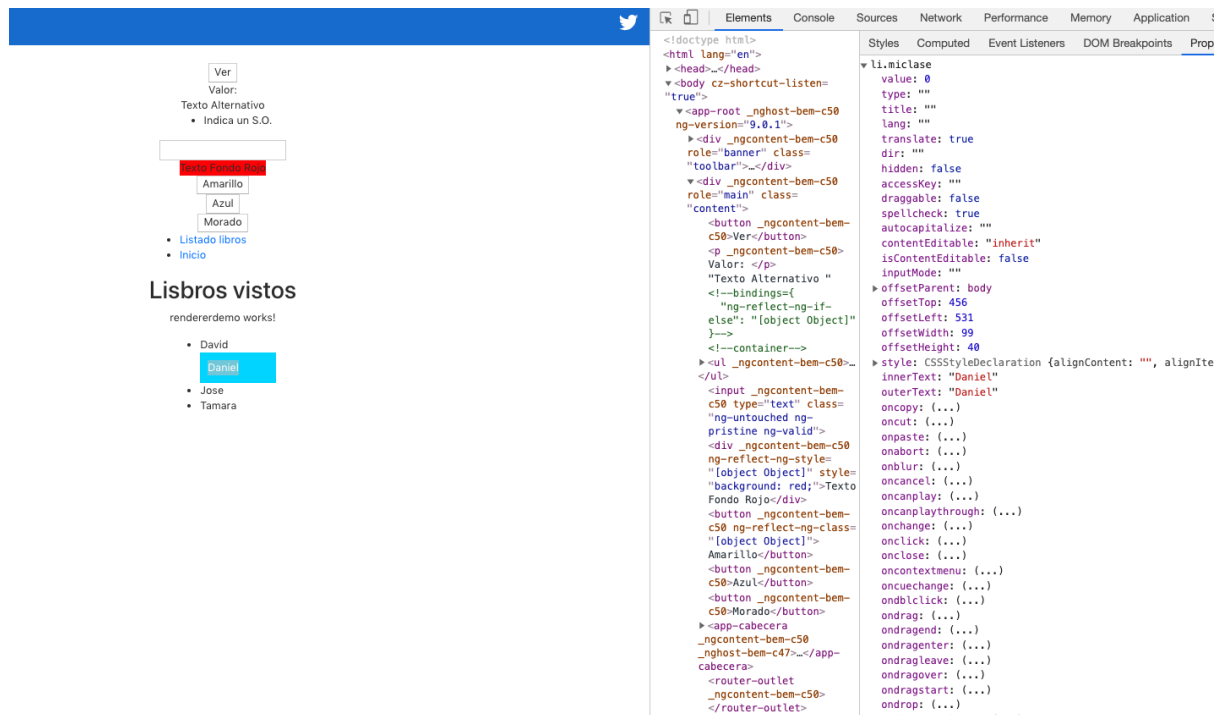
```

De esta sencilla forma, nos vamos guardando el ultimo elemento clickado y cuando se hace click en otro, le quitamos la clase a ese. ¿Fácil no?

Vamos a ver algunas cosas más sobre como controlar el DOM con renderer2.

El Document Object Model es básicamente el conjunto de todos los nodos alineados dentro de él, hablamos de un árbol donde se encuentran todos los elementos interconectados.

Cada uno de estos elementos está asignado con diferentes propiedades. Por ejemplo, mira lo que sucede cuando inspeccionamos un elemento li:



No solo encontramos una larga lista de propiedades, también contaremos con un buen número de eventos. Pues a todo lo que podemos acceder, lo podemos modificar.

Veamos un ejemplo, yo puedo ahcer esto:

```
this.renderer.setAttribute(element, "data-select", "true")
```

Ahora haz click sobre un elemento y mira lo que sucede (inspeccionando)

```
<li _ngcontent-dpg-c57 class="miclase" data-select="true">David</li>
```

Efectivamente! le hemos añadido html. Ahora, incluso con el css puedo hacer:

```
li[data-select="true"]:not(.miclase) {
  background: #BEAAD0
}
```

Con esto, aplicamos una clase adicional al elemento clickado pero que no tenga ya mi clase. Como ves esto nos aporta un gran dinamismo. Ya que podemos añadir nuevos elementos, puedo hacer incluso aparecer un formulario en un evento concreto. Por simplificar, vamos a añadir un elemento html:

```
let nuevoElemento = this.renderer.createElement("span");
this.renderer.setProperty(nuevoElemento, "innerHTML", "✓");
```

```
this.renderer.appendChild( elemen , nuevoElemento );
```

Ejercicio: Ahora tú, cuando selecciono un alumno, crea un botón al lado, que diga "Matricular". Cuando haga click, saca un alert indicando usuario, el nombre del mismo y matriculado.

## 1.13 Publicando nuestra app

Ya tenemos una buena base y es hora de publicar nuestro proyecto. Si vamos a los archivos, veremos que la carpeta pesa bastante y tiene una larga lista de archivos dentro. No todo esto tiene que estar en producción. Para sacar la versión de producción usamos:

**ng build**

**ng build --prod --base-href /url**

<https://github.com/angular/angular-cli/wiki/stories-github-pages>

La magia de todo esto, es que saldrá un proyecto en html aparentemente estático. esto quiere decir que ni siquiera necesito un servidor para tener mi aplicación.

Por ejemplo, si uso AWS puedo poner aquí mi aplicación en un S3 vinculado a un dominio. O si uso Github, puedo tener la app allí y lanzarla al mundo! Veamos cómo hacer esto.

## 2 Angular Nivel 2, pixelshop

Vamos a crearnos un nuevo proyecto llamado pixelshop. Como puedes intuir es una especie de tienda online con la que probar conceptos más avanzados de angular.

Crea un nuevo proyecto e instala bootstrap, ya deberías saber hacer esto perfectamente ;-)  
Si no te acuerdas de la parte de BS no te preocupes, aquí tienes un paso a paso:

### Instalando Bootstrap en Angular

El proyecto está creado con la opción de rutas habilitada.

Dentro de la carpeta raíz del proyecto instalamos Bootstrap y sus dependencias, jQuery y Popper.js:

```
npm install bootstrap jquery popper.js
```

### Configuración de los estilos

Ahora debemos incluir los nuevos estilos dentro del fichero "angular.json" para que nuestra aplicación reconozca los cambios que hemos realizado:



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

En el apartado de scripts agregamos también **Bootstrap** y las dependencias que hemos instalado al inicio.

```
"scripts": [  
  "node_modules/jquery/dist/jquery.slim.min.js",  
  "node_modules/popper.js/dist/umd/popper.min.js",  
  "node_modules/bootstrap/dist/js/bootstrap.min.js"  
],
```

Bootstrap es también un framework de HTML/CSS/JS, es una buena opción cuando queremos montar algo rápido aunque no es tan eficiente como algo hecho a medida. En esta fase de nuestra formación, lo vamos a usar para agilizar el desarrollo de las vistas, responsive... pero en el tercer nivel, MEAN developer, ya pasaremos a los elementos de Angular Material, que si bien son más complejos de usar, también tienen un resultado más óptimo. Si quieres otros recursos para que se encarguen de esta parte, tienes este apartado en la docu:

<https://angular.io/resources?category=development>

Una vez listos, vamos a empezar por crear un modelo. Un modelo es básicamente un archivo donde vamos a poder guardar la estructura de datos de nuestra aplicación. Y, en este caso, vamos a utilizar una interface para darle formato a esa estructura de datos.

**ng generate class models/shop.model**

<https://github.com/bypixelpro/cursoangular9/blob/master/shop.model.ts>

Con esta clase tenemos una estructura de datos, estructura que debemos mantener y para asegurarnos de la correcta implementación, vamos a crearnos una interface.

**ng generate interface interface/Product**

```
export interface Product {  
  title?: string;  
  desc?: string;  
  price?: number;  
  picture?: string;  
}
```

Esta interface, nos indica el tipo de datos que debemos tener. No es una interface estricta ya que la ? al final del nombre nos indica que esa propiedad puede estar o no.

## 2.1 Componentes Stateful vs Stateless

**Componentes Stateful:** Son los componentes que contienen información, operaciones y posiblemente otros componentes Stateless.

**Componentes Stateless:** Son componentes que no contienen un estado y que, usualmente, son componentes "tontos", que lo que hacen es desplegar información.

Vamos a verlo en acción:

### ng generate component stateful

Creamos la lógica que implementa el modelo y la interface:

```
import { Component, OnInit } from '@angular/core';
import { Product } from '../interface/product';
import { Shop } from '../models/shop.model';

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

  shopModel: Shop = new Shop();
  boughtItems: Array<Product>;

  constructor() { }

  ngOnInit(): void {
  }

}
```

Y finalmente montamos una vista:

```
<main role="main" class="container">

  <div class="starter-template">
    <h1>Stateful</h1>
    <p class="lead">Ejemplo de componete stateful.</p>

    <ul class="list-group">
      <li class="list-group-item" *ngFor="let curso of
shopModel.shopItems">{{curso.title}}</li>

    </ul>
  </div>
</main><!-- /.container -->
```

¿Funciona? Bueno, como tenemos BS podemos hacer una versión más vistosa de forma sencilla usando un card:

```
<main role="main" class="container">

  <div class="starter-template">
    <h1>Stateful</h1>
    <p class="lead">Ejemplo de componete stateful.</p>

    <div class="card-deck">
      <div class="card" *ngFor="let curso of
shopModel.shopItems">
        
        <div class="card-body">
          <h5 class="card-title">{{curso.title}}</h5>
          <p class="card-text">{{curso.desc}}</p>
          <p class="card-text">Precio: {{curso.price}}€</p>
          <a href="#" class="btn btn-primary">Comprar</a>
        </div>
      </div>
    </div>
  </div>
</main><!-- /.container -->
```

```

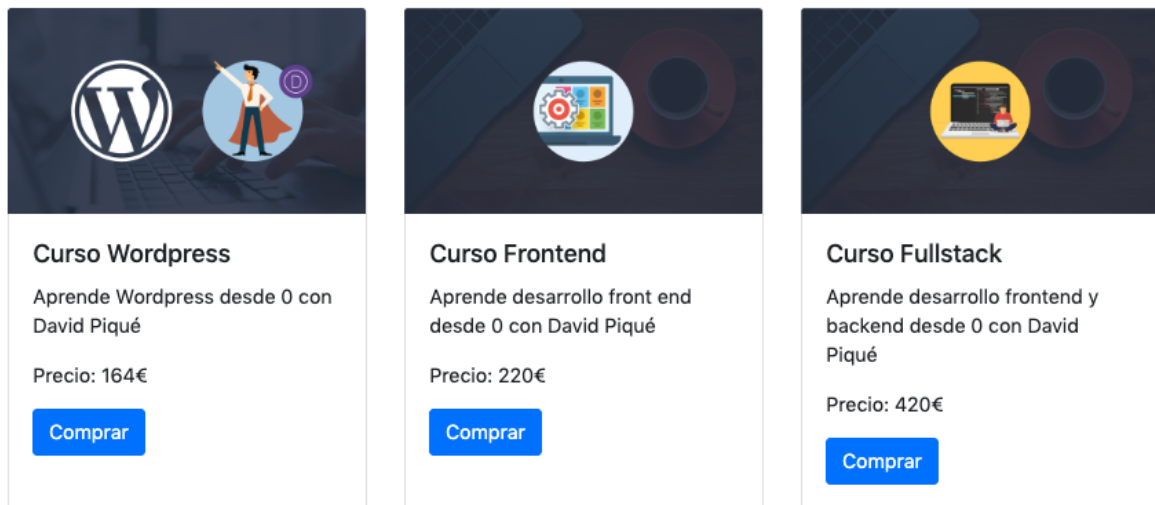
        </div>
    </div>
</div>
</main><!-- /.container -->

```

Y debería quedarnos algo así:

## Stateful

Ejemplo de componente stateful.



Lo siguiente es añadir el click. Como ya debéis saber, para capturar un evento click, usamos ()

```

<a href="#" class="btn btn-primary"
    (click)="clickItem(curso)">Comprar</a>

```

Entonces, una vez que tenemos esto, vamos a ir listando los libros comprados o clickados. En la lógicas, simplemente hacemos un push a un array **boughtItems: Array<Product>;**

```

import { Component, OnInit } from '@angular/core';
import { Product } from '../interface/product';
import { Shop } from '../models/shop.model';

@Component({
  selector: 'app-stateful',
  templateUrl: './stateful.component.html',

```

```
styleUrls: ['./stateful.component.css']
}))
export class StatefulComponent implements OnInit {

shopModel: Shop = new Shop();
boughtItems: Array<Product>;

constructor() { this.boughtItems = [];
}

ngOnInit(): void {

clickItem(curso) {
    this.boughtItems.push(curso);
}
}
```

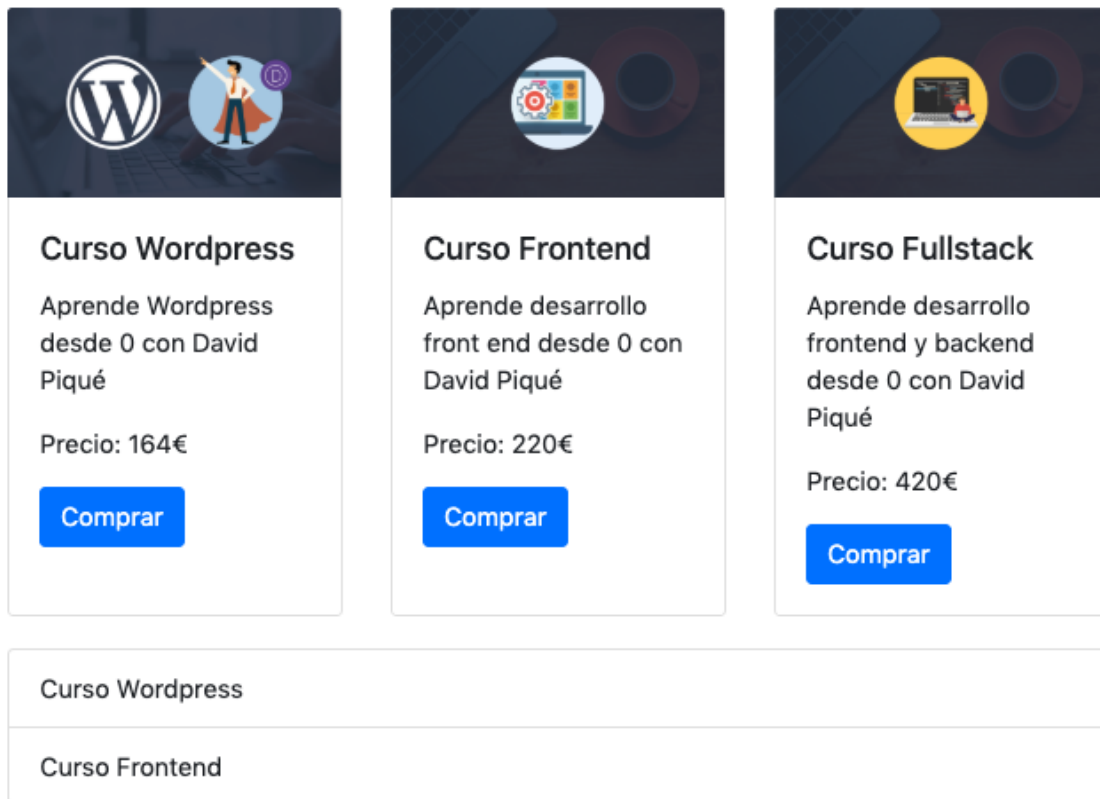
Y en la vista, mostramos ese array:

```
<ul class="list-group">
  <li *ngFor="let boughtItem of boughtItems"
    class="list-group-item">{{boughtItem.title}}</li>
</ul>
```

Listo, ahora cada vez que haces click en un curso, debería mostrarse una lista abajo con lo que vas seleccionando:

## Stateful

Ejemplo de componete stateful.



Fácil verdad? Pues esto es un componente stateful.

Ahora vamos con un stateless. Como ya hemos mencionado, estos tan solo despliegan datos.

Para el servidor y generamos un nuevo componente.

### ng generate component stateless

Ahora, este componente sin estado, se integrará dentro del componente con estado. El componente con estado se encargará de la lógica, y el componente sin estado, simplemente de mostrar datos en la vista.

vamos a llevarnos toda la vista estática del componente con estado a la del componente sin estado:

```

        
        <div class="card-body">
            <h5 class="card-title">{{curso.title}}</h5>
            <p class="card-text">{{curso.desc}}</p>
            <p class="card-text">Precio: {{curso.price}}€</p>
            <a href="#" class="btn btn-primary"
(click)="clickItem(curso)">Comprar</a>
        </div>

```

E integramos el componente sin estado, dentro del con estado, justo donde hemos quitado el código anterior:

```

<main role="main" class="container">

    <div class="starter-template">
        <h1>Stateful</h1>
        <p class="lead">Ejemplo de componete stateful.</p>

        <div class="card-deck">
            <div class="card" *ngFor="let curso of
shopModel.shopItems">
                <app-stateless></app-stateless>
            </div>
        </div>

        <ul class="list-group">
            <li *ngFor="let boughtItem of boughtItems"
class="list-group-item">{{boughtItem.title}}</li>
        </ul>
    </main><!-- /.container -->

```

Quedaría así. El último paso, es hacer que ambos se comuniquen entre sí. En **stateless.component.ts** hacemos lo siguiente:

Lo más importante aquí es el @Input, que hay que importarlo:

```
import { Component, OnInit, Input } from '@angular/core';
import { Product } from '../interface/product';

@Component({
  selector: 'app-stateless',
  templateUrl: './stateless.component.html',
  styleUrls: ['./stateless.component.css']
})
export class StatelessComponent implements OnInit {
  @Input() product: Product;

  constructor() { }

  ngOnInit(): void {
  }
}
```

Ahora, el producto ya no es curso, es product, así que debemos cambiar la vista de el componente sin estado:

```

<div class="card-body">
  <h5 class="card-title">{{product.title}}</h5>
  <p class="card-text">{{product.desc}}</p>
  <p class="card-text">Precio: {{product.price}}€</p>
  <a href="#" class="btn btn-primary"
    (click)="clickItem(curso)">Comprar</a>
</div>
```



Y permitir que el padre le pase correctamente los datos, en **stateful.component.html**:

```
<app-stateless [product]="curso"></app-stateless>
```

En este punto, ambos componente hablan entre si y debería mostrarse igual que antes. Es cierto que el botón no funcionará. ya que el componente sin estado no tiene ese método, solo recibe desde su padre, no te preocupes, lo vamos a ir solucionando. Vamos a la lógica del stateless y añadimos el texto de comprar y el precio.

En mi caso, cuando un alumno compra un curso, ya no lo puede volver a comprar, entonces vamos a desactivar el botón. Para ello tendremos 2 métodos:

**Matricularse()**: Desactivara el botón (BS tiene una clase para eso) y cambiará el texto  
**isDisabled()**: Solo retorna true o false.

Como el código empieza ser extenso, os paso enlaces a los archivos.

<https://github.com/bypixelpro/cursoangular9/blob/master/stateless.component.ts>

Y su vista correspondiente:

<https://github.com/bypixelpro/cursoangular9/blob/master/stateless.component.html>

Y listo, con esto tienes el componente stateless funcionando, solo se puede matricular una vez, luego el botón se desactiva.

Importante, lo que tenemos en la app en este momento es x1 componente stateful y x3 stateless. El padre, inyecta información en los stateless. El siguiente desafío es hacer que el hijo, envíe información al padre. Esto lo conseguiremos mediante eventos.

El hijo, notifica al padre mediante un evento y en esta arquitectura es siempre el padre el que realiza la acción.

## 2.2 El maravilloso mundo de los eventos

En el componente sin estado, vamos a añadir un emisor de eventos:

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

```
@Output() cursomatriculado: EventEmitter<Product> = new  
EventEmitter();
```

Esto es un emisor de eventos, ahora tenemos que despacharlo y lo hacemos en el método matricularse() Que quedaria asi:

```
matricularse() {
```

```
this.disable = true;
this.matricula = '¡Matriculado!';
this.cursomatriculado.emit(this.product);
}
```

Finalmente, el padre, recibe ese evento, vamos a la vista del componente stateful. Para capturar el evento usamos los parentesis:

```
<app-stateless [product]="curso"
(cursomatriculado)="cursoMatriculado($event)"></app-stateless>
```

En la lógica del componente stateful implementamos este evento, para listar los cursos en los que se ha matriculado el usuario.


```
cursoMatriculado(_event: Product) {
  this.clickItem(_event);
}
```

Listo! Ya tienes una arquitectura padre hijo, stateful stateless. Vamos a terminar diseñando un vistoso carrito de compra.

**Ejercicio tú solo.** Muestra al final del detalle, el importe total que ha de pagar el usuario.

# Stateful

Ejemplo de componente stateful.




**Curso Wordpress**

Aprende Wordpress desde 0 con David Piqué

Precio: €164.00

**Matricularse**




**Curso Frontend**

Aprende desarrollo front end desde 0 con David Piqué

Precio: €220.00

**¡Matriculado!**



**Curso Fullstack**

Aprende desarrollo frontend y backend desde 0 con David Piqué

Precio: €420.00

**¡Matriculado!**

Información matrícula
Curso Frontend <b>€220.00</b>
Curso Fullstack <b>€420.00</b>
<b>Total: €640.00</b>

Repositorio resuelto:

<https://github.com/bypixelpro/pixelshop/tree/solucioncarrito>

## 2.3 Control de componentes anidados con ViewChild

Vamos a mostrar una ventana de confirmación cuando llega el momento de pagar.

Genera un nuevo componente con: **ng generate component confirm**

Este componente, lo vamos a poner dentro del componente padre:

```
<app-confirm></app-confirm>
```

Luego en la vista de este componente vamos a añadir un modal de Bootstrap:

```
<!-- Button trigger modal -->
```

```
<button type="button" class="btn btn-primary" data-toggle="modal" data-target="#exampleModal" [disabled]="isDisabled">
```

```

    Confirmar compra
</button>

<!-- Modal -->
<div class="modal fade" id="exampleModal" tabindex="-1"
role="dialog" aria-labelledby="exampleModalLabel"
aria-hidden="true">
  <div class="modal-dialog" role="document">
    <div class="modal-content">
      <div class="modal-header">
        <h5 class="modal-title" id="exampleModalLabel">Confirmación
de matriculación</h5>
        <button type="button" class="close" data-dismiss="modal"
aria-label="Close">
          <span aria-hidden="true">&times;</span>
        </button>
      </div>
      <div class="modal-body">
        ...
      </div>
      <div class="modal-footer">
        <button type="button" class="btn btn-secondary"
data-dismiss="modal">Cerrar</button>
        <button type="button" class="btn
btn-primary">Confirmar</button>
      </div>
    </div>
  </div>
</div>

```

Ya tienes un botón, con una ventana modal que confirma la compra. El botón por defecto, ha de estar desactivado, ya que si en un principio no hay cursos, no se debería poder comprar. Por eso tenemos esta etiqueta en el botón:

```
[disabled]="isDisabled"
```

Esta etiqueta es de BS y su valor lo controlamos desde el controlador (confirm.component.ts con esta propiedad:

```
isDisabled: boolean;
```

```
constructor() {  
    this.isDisabled = true;  
}
```

Ahora, lo esencial de este ejercicio, es hacer que desde el componente stateful, podamos controlar cuando se ha de activar el botón. Una vez se añade un curso, activamos el botón.

Ves al **stateful.component.ts**:

#### 1 Importamos ViewChild

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

#### 2 Lo inicializamos

```
@ViewChild(ConfirmComponent, {static: false})
```

#### 3 Lo instanciamos en una propiedad

```
confirmChild: ConfirmComponent;
```

#### 4 Modificamos su valor desde el método que añade el curso al carro

```
cursoMatriculado(_event: Product) {  
    this.clickItem(_event);  
    this.confirmChild.isDisabled = false;  
}
```

Resultado final:

```
import { ConfirmComponent } from '../confirm/confirm.component';  
import { Component, OnInit, ViewChild } from '@angular/core';  
import { Product } from '../interface/product';  
import { Shop } from '../models/shop.model';
```

```

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

  shopModel: Shop = new Shop();
  boughtItems: Array<Product>;

  @ViewChild(ConfirmComponent, {static: false})
  confirmChild: ConfirmComponent;

  constructor() { this.boughtItems = [];
  }

  ngOnInit(): void {
  }

  clickItem(curso) {
    this.boughtItems.push(curso);
  }

  cursoMatriculado(_event: Product) {
    this.clickItem(_event);
    this.confirmChild.isDisabled = false;
  }

  finalPrice() {
    if (this.boughtItems) {
      return this.boughtItems.reduce(
        (prev: number, item: Product) => prev + item.price, 0
      );
    }
  }
}

```

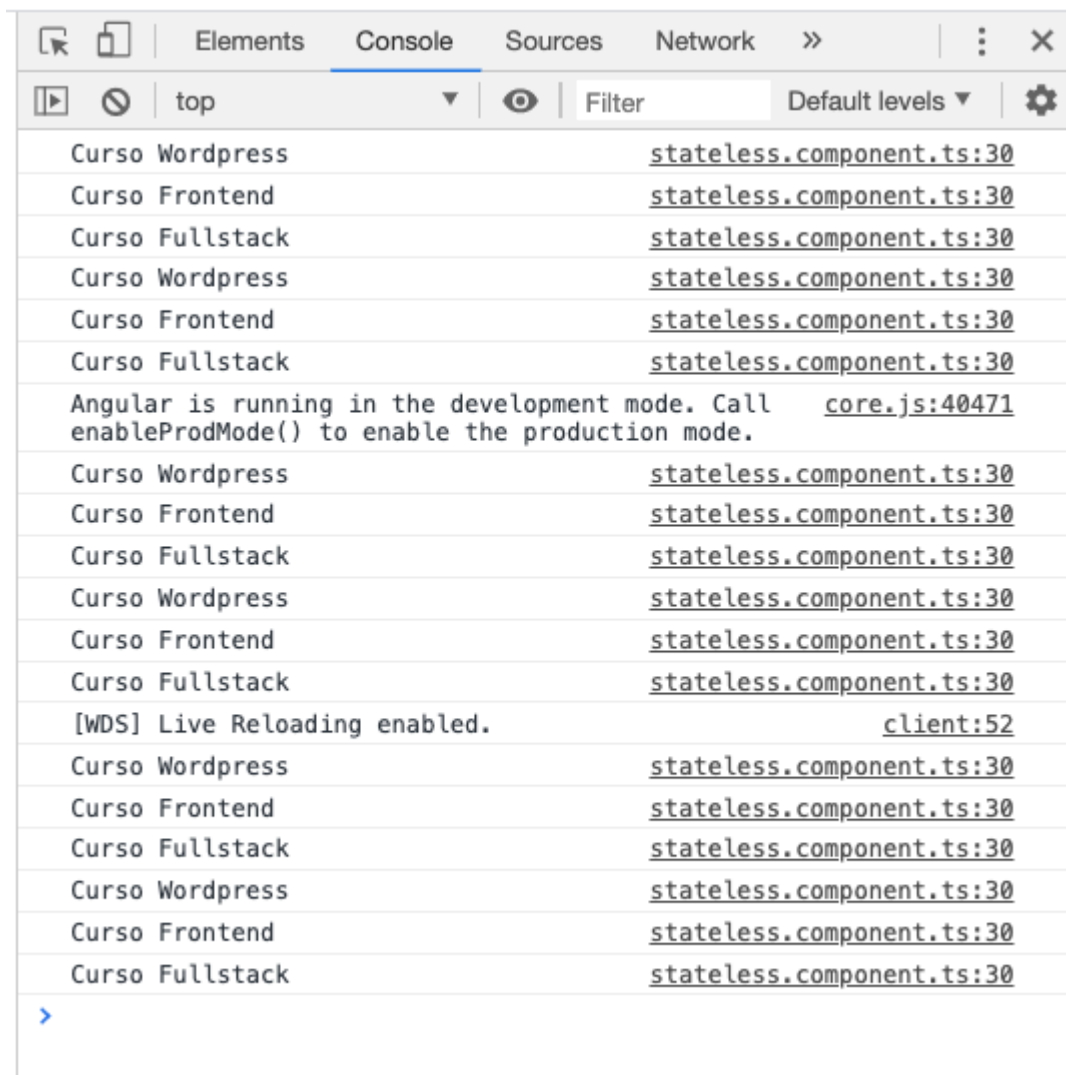
```
}  
}
```

## 2.4 Detectando cambios y mejorando la eficiencia

En nuestro componente stateless, vamos a hacer una prueba para que veas lo que está haciendo angular por detrás y como esta arquitectura, puede terminar influenciando al rendimiento. Vamos a: **stateless.component.ts**

```
isdisabled() {  
  console.log(this.product.title);  
  return !!this.disable;  
}
```

Hacemos un console log y mira lo que pasa, mira cuantas veces se instancia...



Ahora en este mismo componente, vamos a implementar una política de detección de cambio, en push en lugar de como esta por defecto y verás que mejora significativamente:

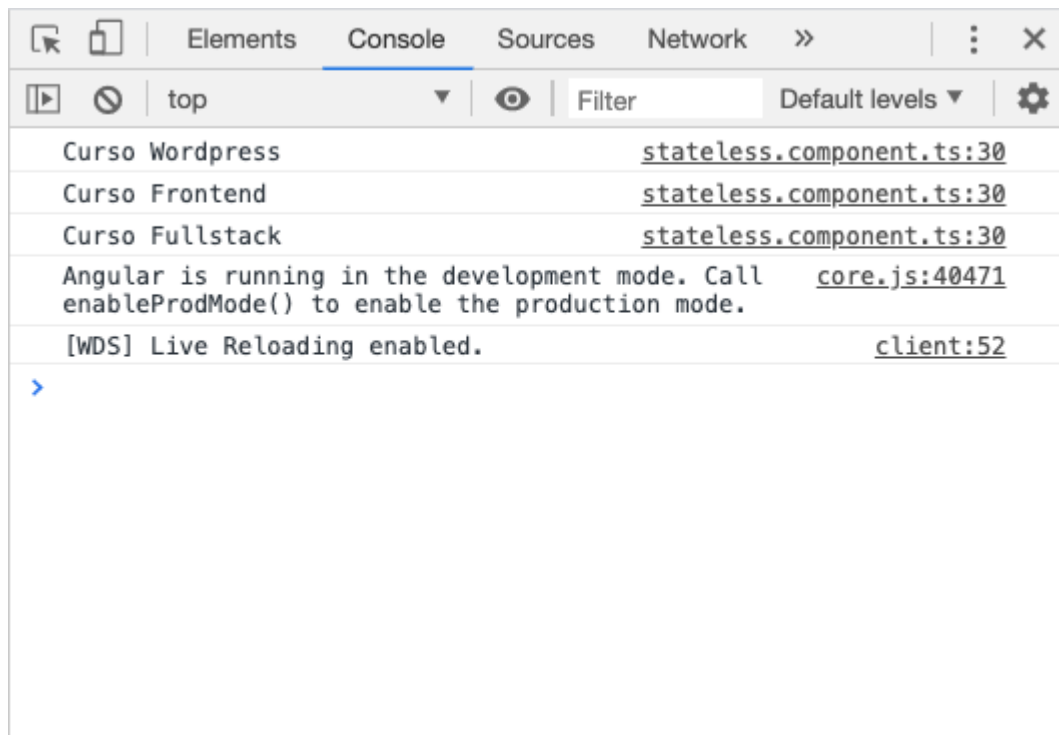
Importamos:

```
import { Component, OnInit, Input, Output, EventEmitter,
ChangeDetectionStrategy } from '@angular/core';
```

y añadimos al decorador:

```
@Component({
  selector: 'app-stateless',
  templateUrl: './stateless.component.html',
  styleUrls: ['./stateless.component.css'],
  changeDetection: ChangeDetectionStrategy.OnPush
})
```





Mejor no?

## 2.5 Eventos y ciclo de vida.

## 2.6 OnDestroy

También tenemos el evento `NgOnDestroy`, hay muchas situaciones dentro de Angular que pueden invocar un componente a ser destruido.

Uno muy frecuente, es el de la suscripción, veamos cómo destruir los cualquier llamada a la memoria con el `ngOnDestroy` y vamos a hacer que, en el momento en que nuestro componente salga del flujo de la lógica de nuestra aplicación, se destruya por completo, no solo el componente, sino también todas las llamadas y todos los punteros de memoria que esté ocupando.

Empezamos en este punto:

<https://github.com/bypixelpro/pixelshop/tree/json>

y añadiremos esto:

<https://github.com/bypixelpro/cursoangular9/blob/master/ondestroy.ts>

Esta implementación requiere de varios pasos:

1. Importar el `ondestroy`

2. Extender la clase
3. Añadir el método ngOnDestroy
4. Suscribirse y desuscribirse

```
ngOnDestroy() {  
  this.shopSubscription.unsubscribe();  
}
```

El resultado final lo puedes ver aqui:

<https://github.com/bypixelpro/pixelshop/tree/jsonterminado>

## 2.7 OnChanges

Hay un detector de cambios en angular que funciona realmente bien y podemos saber todo lo que sucede en nuestra app, de cara a la interacción del usuario y los Inputs, vamos a ver como se implementa.

1. Creamos un componente para ver el estado de la cesta **ng generate component status-cart**
2. Lo incorporamos al componente stateful
3. Vamos a incluir en el componente del status, inputs que nos permitan recibir información

Nos quedaria algo asi:

```
import { Component, EventEmitter, Input, OnChanges, OnInit, SimpleChanges } from  
'@angular/core';  
import { Product } from '../interface/product';
```

```
@Component({  
  selector: 'app-status-cart',  
  templateUrl: './status-cart.component.html',  
  styleUrls: ['./status-cart.component.css']  
})  
export class StatusComponent implements OnInit, OnChanges {
```

```
  @Input() price: number;  
  @Input() shopModel: Array<Product>;  
  @Input() add: EventEmitter<null> = new EventEmitter();
```

```
  constructor() { }
```

```
  ngOnInit(): void {  
  }
```

```

ngOnChanges(changes: SimpleChanges) {
  console.log(changes);
}

confirm() {
  this.add.emit();
}

}

```

En la implementación del componente con estado, deberíamos pasar los datos que esperamos en los inputs:

```

<app-status-cart
  [price]="finalPrice()" [shopModel]="shopModel.shopItems"
  (add)="onConfirm()"
>
</app-status-cart>

```

Ahora, al componente del status, le añadimos el método onchanges, éste solo puede incluirse en componentes hijos. Este solo rastrea los cambios que se realizan en los **inputs**.

Finalmente, este sería el componente stateful:

```

import { ConfirmComponent } from '../confirm/confirm.component';
import { Component, OnInit, ViewChild, OnDestroy } from
 '@angular/core';
import { Product } from '../interface/product';
import { Shop } from '../models/shop.model';
import { HttpClient, HttpResponse } from '@angular/common/http';
import { Subscription } from 'rxjs';

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

```

```

export class StatefulComponent implements OnInit, OnDestroy {

  errorHttp: boolean;
  shopModel;
  boughtItems: Array<Product>;

  @ViewChild(ConfirmComponent, {static: false})
  confirmChild: ConfirmComponent;

  private shopSubscription: Subscription;

  constructor(private http: HttpClient) {
    this.boughtItems = [];
    this.shopModel = {shopItems: []};
  }

  ngOnInit(): void {
    this.shopSubscription =
    this.http.get('assets/cursos.json').subscribe(
      (respuesta: Response) => { this.shopModel.shopItems =
respuesta; },
      (respuesta: Response) => { this.errorHttp = true }
    )
  }

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

  clickItem(curso) {
    this.boughtItems.push(curso);
  }

  cursoMatriculado(_event: Product) {

```

```

    this.clickItem(_event);
    this.onConfirm();
    this.confirmChild.isDisabled = false;
  }
  onConfirm() {
    alert('Has añadido un nuevo curso');
  }


  finalPrice() {
    if (this.boughtItems) {
      return this.boughtItems.reduce(
        (prev: number, item: Product) => prev + item.price, 0
      );
    }
  }
}

```

Solo hemos puesto un método, con un alert, para confirmar elementos añadidos al carro. pero lo interesante de todo esto, es cuando hacemos debug:

## Stateful

Ejemplo de componente stateful.




**Curso Wordpress**

Aprende Wordpress desde 0 con David Piqué

Precio: €164.00

¡Matriculado!




**Curso Frontend**

Aprende desarrollo front end desde 0 con David Piqué

Precio: €220.00

Matriculase



**Curso Fullstack**

Aprende desarrollo frontend y backend desde 0 con David Piqué

Precio: €420.00

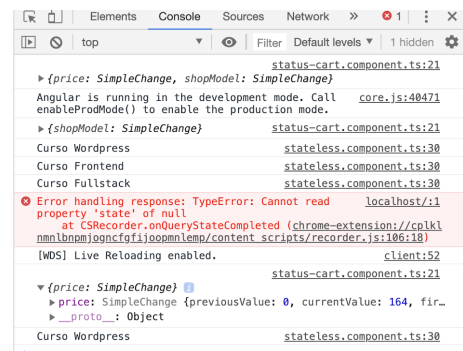
Matriculase

Información matrícula

Curso Wordpress **€164.00**

**Total: €164.00**

Confirmar compra



```

status-cart.component.ts:21
    {price: SimpleChange, shopModel: SimpleChange}
Angular is running in the development mode. Call enableProdMode() to enable the production mode.
    {shopModel: SimpleChange}
Curso Wordpress
Curso Frontend
Curso Fullstack
Error handling response: TypeError: Cannot read property 'state' of null
    at CSRecorder.onQueryStateCompleted (chrome-extension://colklmmmlbnpmjogncifgjjoommlmp/content_scripts/recorder.js:106:18)
[WDS] Live Reloading enabled.
    {price: SimpleChange}
    price: SimpleChange {previousValue: 0, currentValue: 164, fir...
    __proto__: Object
Curso Wordpress

```

Fíjate que **SimpleChange**, nos indica lo que está pasando con el **precio**!

Se te ocurre alguna aplicación práctica?

## 2.8 OnDoCheck

La principal diferencia con el anterior, es que OnChanges, mira la modificación de los valores de los Inputs y OnDoCheck, se centra en modificaciones del DOM. Este rastrea toda interacción en la aplicación pero tiene un coste grande en el uso de memoria, ya que como acabamos decir, rastrea todo.

Su implementación es muy sencilla, en el componente de estatus que hemos hecho, haz lo siguiente:

```
export class StatusCartComponent implements OnInit, OnChanges,
DoCheck {
```

y luego:

```
ngDoCheck() {
  console.log('Interacción usuario');
}
```

**Importante, no se recomienda tener ambos a la vez!**

## 2.9 Mouse Events

Algo que sí es muy recomendable, es capturar los eventos del ratón o puntero, vamos a ver como hacerlo. Ya sabes cómo capturar el click normal con (**click**) ahora vamos a realizar lo mismo con el click derecho, para ello usamos (**contextmenu**)

Ejemplo:

```

```

## 2.10 Keyboard Events

Otra cosa que nos permite Angular, es capturar eventos de teclado. Esto se puede aplicar a campos de formulario, donde podremos ver que está introduciendo el usuario dentro de un campo cuando el foco está sobre este, e incluso responder a teclas específicas. Veamos un ejemplo.

**Vista:**

```
<input type="text" (keydown)="onKeyboard($event)"/>
```

**Componente:**

```
onKeyboard(_event) {
  console.log(_event);
```

```
}
```

Si examinamos el objeto en el inspector, veras una propiedad que es:

```
KeyboardEvent {isTrusted: t  
  "KeyH", location: 0, ctrlKe  
    isTrusted: true  
    key: "H"  
    code: "KeyH"  
    location: 0
```

Aquí podemos ver la tecla pulsada.

**Ejercicio**, si el usuario presiona “enter” saca un alert.

```
onKeyboard(_event) {  
  if (_event.key === 'Enter' ) {  
    this.onConfirm();  
  }  
}
```

Ya hemos visto cómo escuchar los eventos del teclado de un usuario cuando el foco está en un campo input. Ahora vamos a ver cómo escucharlos de manera global:

```
onGlobalKeyboard() {  
  document.addEventListener('keypress', (eventoGlobal) => {  
    this.onKeyboard(eventoGlobal);  
  });  
}
```

¿Se te ocurre cómo implementarlo?

Efectivamente, en el ngOnInit

```
this.onGlobalKeyboard();
```

Importante, esta práctica, nos puede llevar a perdidas de memoria impiortantes, ya que el método en el oninit, se quedará escuchando lo que pase en el teclado durante toda la vida de la aplicaión, asi que es importante una vez el componente cierre, destruirlo, para ello:

```
ngOnDestroy() {  
  document.removeEventListener('keypress', this.onKeyboard);  
  this.shopSubscription.unsubscribe();  
}
```

## 2.11 Formularios

Para practicar con los formularios, vamos a crearnos un nuevo proyecto. Yo para este, como es algo sencillo, voy a usar chota.

Importaremos el formsmodule en el **app.module.ts**:

```
import { FormsModule } from '@angular/forms';
...
imports: [
  BrowserModule, FormsModule
],
```

Con esto, ya podemos trabajar con los elementos de formulario de Angular. Ahora nos vamos a crear un componente:

### **ng generate component formuarioprimer**

En la vista del componente, lo siguiente será conectar la vista a angular, instanciando en ngform. Puedes empezar con algo básico:

```
<section id="forms">
  <header>
    <h1>Form elements</h1></header>
  <form #formuarioprimer="ngForm">
    <fieldset id="forms__input">
      <legend>Input fields</legend>
      <p>
        <label for="user">Usuario</label>
        <input name="user" id="user" type="text" placeholder="Su
nombre de usuario" required ngModel #user="ngModel">
      </p>
    </fieldset>
  </form>
</section>

{{formuarioprimer.value | json}}
```

Vamos a explicarlo, para que el formulario sea un formulario de angular, usamos:

```
<form #formuarioprimer="ngForm">
```

Luego en el campo en cuestión que estamos manejando, o escuchando, usamos:

```
<input name="user" id="user" type="text" placeholder="Su nombre de
usuario" required ngModel #user="ngModel">
```

Este tipo de comunicación es bidireccional, es decir, la vista habla con el módulo y el módulo con la vista. Para verlo podemos hacer lo siguiente:



```
{{formularioprimer.value | json}}
```

Para enviar esto, en lugar de usar el método tradicional, de un campo **action** que redirige a un archivo externo de php, lo hacemos de la siguiente manera. Lo primero es capturar el evento click, cuando el usuario presiona el botón.

Añadimos un botón de submit y un nuevo campo al usuario:

```
<section id="forms">
  <header>
    <h1>Form elements</h1></header>
  <form #formularioprimer="ngForm"
    (ngSubmit)="onSubmit(formularioprimer)">
    <fieldset id="forms__input">
      <legend>Input fields</legend>
      <p>
        <label for="user">Usuario</label>
        <input name="user" id="user" type="text" placeholder="Su
nombre de usuario" required ngModel #user="ngModel">
      </p>
      <p>
        <label for="password">Password</label>
        <input name="password" id="password" type="password"
required ngModel #user="ngModel">
      </p>
      <p><button class="button primary">Submit</button></p>
    </fieldset>
  </form>
</section>
```

Y capturamos los datos en el componente:

```
onSubmit(_datosForm) {
  console.log(_datosForm.value);
}
```

Vamos a ver como validar los datos, lo primero es añadir la etiqueta **novalidate** al formulario para evitar conflictos con la validación de HTML. Una tarea sencilla es, ponemos los dos campos como requeridos. Una vez lo están, no mostramos el botón hasta que no están cumplimentados. En angular, mediante [ ] tenemos acceso a las propiedades de una etiqueta en concreto.

```

<section id="forms">
  <header>
    <h1>Form elements</h1></header>
    <form #formularioprimer="ngForm" (ngSubmit)="onSubmit(formularioprimer)"
novalidate>
      <fieldset id="forms__input">
        <legend>Input fields</legend>
        <p>
          <label for="user">Usuario</label>
          <input name="user" id="user" type="text" placeholder="Su nombre de
usuario" required ngModel #user="ngModel">
        </p>
        <p>
          <label for="password">Password</label>
          <input name="password" id="password" type="password" required ngModel
#user="ngModel">
        </p>
        <p><button class="button primary" type="submit"
[disabled]="!formularioprimer.valid">Submit</button></p>
      </fieldset>
    </form>
    {{formularioprimer.valid | json}}
  </section>

```

## Formularios Reactivos

Esto es un formulario convencional, ahora vamos a hacer uno reactivo, en este planteamiento, es el componente el que controla la vista. Para ello importamos el módulo.

```

import { FormsModule, ReactiveFormsModule } from '@angular/forms';
...
imports: [
  BrowserModule, FormsModule, ReactiveFormsModule
],

```

En el componente del formulario **formularioprimer.component.ts** importamos lo siguiente:

```

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

```

En en la clase y en el constructor añadimos una variable pública y otra privada:

```
formulario: FormGroup;  
constructor(private FormBuilder: FormBuilder) { }
```

Lo preparamos en el evento ngOnInit:

```
ngOnInit(): void {  
    this.formulario = this.formBuilder.group({  
        user: 'Miusuario',  
        password: ''  
    });  
}
```

Y modificamos la vista:

```
<section id="forms">  
    <header>  
        <h1>Form elements</h1></header>  
    <form [formGroup]="formulario" (ngSubmit)="onSubmit(formulario)"  
novalidate>  
        <fieldset id="forms__input">  
            <legend>Input fields</legend>  
            <p>  
                <label for="user">Usuario</label>  
                <input formControlName="user">  
            </p>  
            <p>  
                <label for="password">Password</label>  
                <input formControlName="password">  
            </p>  
            <p><button class="button primary" type="submit"  
[disabled]="!formulario.valid">Submit</button></p>  
        </fieldset>  
    </form>
```

```
</section>
```

Como podrás ver nuestro formulario se mantiene, pero tenemos que añadir las validaciones, para ello importamos:

```
import { FormsModule, ReactiveFormsModule, Validators } from
  '@angular/forms';
```

```
ngOnInit(): void {
  this.formulario = this.formBuilder.group({
    user: ['David', [Validators.required,
Validators.minLength(2)]],
    password: ['', Validators.required]
  });
}
```

Como ves, tenemos un array con las distintas validaciones para cada campo. Aquí podemos incluso añadir expresiones regulares para las validaciones más complejas. En la vista, validamos de este modo:

```
<span
*ngIf="formulario.get('user').hasError('required')">Este campo es
requerido</span>
{{formulario.controls.user.errors | json}}
```