



Angular S2: Templates & Directivas | Inputs & Outputs

Después de esta lección podrás:

1. Conocer la sintaxis de Angular.
2. Uso y sintaxis de atributos dinámicos.
3. Conocer y aplicar las directivas de Angular en un proyecto.
4. Establecer comunicación entre componentes.
5. Entender la estructura de un proyecto Angular.
6. Trabajar con soltura usando Inputs & Outputs.

Templates & Directivas

Template Syntax

En la anterior lección vimos que podemos interpolar variables en nuestro código así:

```
<p>the variable name exists and the value is {{ name }}</p>
```

Y tener la variable de nombre en nuestro componente hará que aparezca en nuestra aplicación:

```
export class ExampleComponentComponent implements OnInit {  
  name: string = 'Student';  
  
  constructor() {}  
  
  ngOnInit() {}  
}
```

¿Qué pasa con los atributos como las fuentes de imágenes? Podemos interpolar el valor:

```
<img src={{imgSource}} alt="image_alt">
```

ngModel:

Implementa un mecanismo de *binding* bi-direccional. El ejemplo típico es con el elemento HTML `<input>`, donde asigna la propiedad *value* a mostrar y además responde a eventos de modificación. Algo que veremos más adelante en el día de hoy.

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

ngClass:

Esta directiva permite añadir/eliminar varias clases a un elemento de forma simultánea y dinámica. El ejemplo es algo más complejo.

```
<div [ngClass]="{'pending': !isDone, 'done': true, 'other-class-name': false}">Este Todo es importante y está pendiente</div>
```

ngStyle:

De forma análoga a `ngClass`, esta directiva te permite asignar varios estilos inline a tu elemento. Veamos:

```
<!--in template-->
<div [ngStyle]="{'background': 'red'}"></div>
```

Eventos

Podemos usar un botón como este:

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

Y crear el método en el archivo `component.ts`:

```
onButtonClick() : void {
  console.log('clicked!')
}
```

Veréis que al clicar sobre el botón se imprimirá el mensaje tantas veces como clickemos sobre este. De este modo podéis observar el funcionamiento de los eventos en angular de una manera sencilla.

Directivas

Las directivas corresponden a elementos en el HTML que permiten añadir, manipular o eliminar elementos del DOM. Estas directivas son fácilmente reconocibles debido a que están precedidas por un *asterisco* * seguido del nombre de la directiva. Por ejemplo:

```
<div *ngIf="condition or variable">
  <p>Contition is true or variable is true</p>
</div>

//Acción de la directiva es *ngIf
```

Es como un if que engloba todo el contenido dentro del **<div>** contendor.

¿Qué sucede si **no** queremos que aparezca el bloque dentro de nuestro **DOM**? Podemos usar la etiqueta **<ng-template>** durante el proceso de transpilación de Angular CLI. Por ejemplo:

```
<ng-template [ngIf]="condition or variable">
  <div>
    <p>Contion or Variable is true and draw on DOM</p>
  </div>
</ng-template>
```

Al utilizar una directiva estructural, el transpilador agrega un elemento **<ng-template>** que envuelve el elemento y ocurren dos cosas:

- La directiva (**ngIf** en este caso) pasa al elemento padre **<ng-template>** .
- Los elementos hijos (incluyendo todos sus atributos), pasan a estar dentro de **ng-template** .

Tipos de directivas

En Angular existen dos tipos de directivas:

1. **Directivas de componente:** es aquella que se utiliza en un punto del HTML de manera "nativa" por lo tanto sería una directiva con un template.
2. **Directivas de atributo:** con esta cambiamos la apariencia o comportamiento de un elemento.

***Importante!** solamente se puede aplicar una directiva por bloque es decir:

```
<!-- WORKS -->
<p *ngIf="condition or variable"> Una directiva en bloque</p>

<!-- NOT WORKS -->
<p *ngIf="condition or variable" *ngFor="let element of myArray"> Dos directivas</p>

<!--SOLUTION -->
<ng-template [ngIf]="condition or variable">
  <div *ngFor="let element of myArray">
    <p>Dos directivas y no son dos divs</p>
  </div>
</ng-template>
```

Pero bueno, estamos usando ejemplos de directivas y no os hemos dicho que Angular posee 2 directivas estructurales:

- ***ngIf**: Permite mostrar / ocultar elementos del DOM.
- ***ngFor**: Permite ejecutar bucles sobre elementos del DOM.

*ngIf

Esta directiva toma una expresión booleana y hace que todo elemento contenido del DOM aparezca o desaparezca dada esa condición. Por ejemplo:

```
<!-- BOOLEAN CONTION -->
<p *ngIf="condition or variable">Una directiva en bloque</p>
```

Un ejemplo de uso:

```
<!-- Dado let avengers: boolean = true. El elemento va a aparecer -->
<p *ngIf="avengers">Avengers exists</p>

<!-- Dado let avengers: boolean = false. El elemento no va a aparecer -->
<p *ngIf="avengers">Avengers not exist</p>
```

***Importante!** si recordáis los operadores lógicos que os explicamos en el prework se pueden usar aquí sin problema, os recuerdo algunos como && or ||.

*ngFor

Si recordáis el bucle `for` y sus variantes, no deberíais tener problema alguno para usar esta directiva ya que no deja de ser un `for of`. Para realizar este bucle hay que:

1. Declarar la variable que contiene el valor de la iteración.
2. Utilizar `of`.
3. La variable / Array que vamos a iterar.

También se puede añadir aunque no es necesario:

1. Índice de la iteración.
2. Imprimir el valor de la iteración.

Estructura de la directiva:

```
avengers = ['Spiderman', 'Iron-man', 'Hulk', 'Thor']
```

```
<!--Optional part let i = index" -->
<ul>
  <li *ngFor="let avenger of avengers; let index = index">
    Nombre del Vengador número {{index}} de nombre {{avenger}}
  </li>
</ul>
```

Inputs & Outputs

Después de haber creado varios componentes hemos llegado al punto en el que querríamos que ciertos componentes compartiesen información, de tal manera que cualquier cambio en uno afecte a otro. Por ello os vamos a explicar la comunicación entre componentes, de padre a hijo o de hijo a padre a través de los **Inputs & Outputs**. 🍌

Inputs

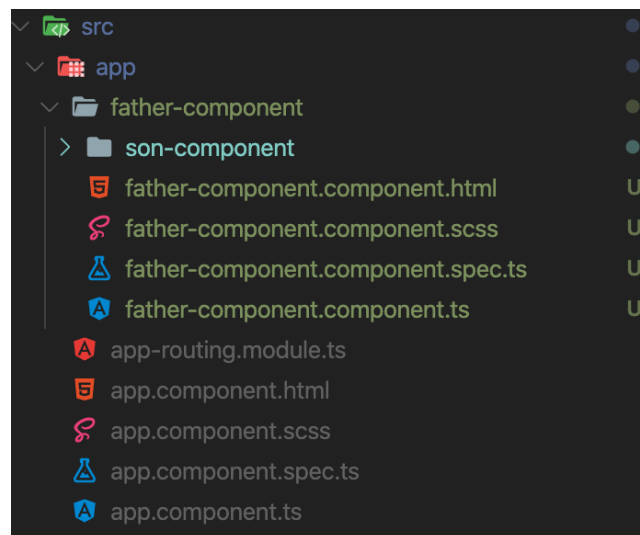
Los inputs indican que les entra un valor desde otro punto de nuestra aplicación, en este caso de un componente padre. Para entenderlo mejor empezaremos creando un nuevo proyecto

```
ng new communication-app
```

Una vez creada nuestra aplicación debemos generar dos componentes, un componente padre y un componente hijo. Tal y como vemos aquí:

```
cd communication-app  
  
cd src/app  
  
ng generate component father-component  
  
cd father-component  
  
ng generate component son-component
```

De este modo tendríamos la siguiente estructura en nuestro proyecto:



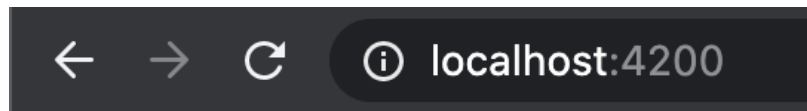
Lo primero como siempre es limpiar el **app.component.html**:

```
<app-father-component></app-father-component>
<router-outlet></router-outlet>
```

Y desde nuestro componente padre instanciaremos el componente hijo:

```
<p>father-component works!</p>
<app-son-component></app-son-component>
```

Ahora si inicializamos nuestra aplicación con **ng-serve** veremos:



father-component works!

son-component works!

Hasta aquí todo sencillo y no hay novedad respecto a clases anteriores. Pero ha llegado el momento de utilizar un poco de la magia que nos da Angular. Para ello vamos a definir un input de texto en el componente padre que cuando escribamos sobre el envíe la info al componente hijo que irá mostrando el texto escrito a tiempo real. Allá vamos!

En el **father-component.component.ts**:

```
export class FatherComponentComponent {
  //declaramos nuestra variable
  inputText: string = '';

  // Función cada vez que pulsamos una tecla.
  keyUp(letra: string) {
    this.inputText = letra;
  }
}
```

Y en el **father-component.component.html**:

```
<div>
  <h1>Father Component</h1>
  <!-- Input sobre el que escribimos y la función declarada en el .ts -->
  <input #fatherInput (keyup)="keyUp(fatherInput.value)"/>
  <div>
    <!-- pasamos al hijo el valor de la variable inputText -->
    <!-- El valor de la izquierda es el que recibe -->
    <app-son-component [inputText]="inputText"></app-son-component>
  </div>
</div>
```

De este modo si ejecutamos el `ng-serve` para arrancar nuestra aplicación observaremos que nos da un error indicando que **inputText** no es conocido por el componente `app-son-component`. Para ello aplicaremos la magia de Angular usando el decorador `@Input`.

En nuestro **son-component.component.ts**:

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

@Component({
  selector: 'app-son-component',
  templateUrl: './son-component.component.html',
  styleUrls: ['./son-component.component.scss']
})
export class SonComponentComponent {
  // Recibe inputText desde la template del padre
  @Input() inputText: string;
}
```

Y desde **son-component.component.html** disponemos de esa variable que tiene un efecto reactivo, cuando cambia en el padre afecta a el hijo.

```
<div>
  <h1>Son Component</h1>
  <p>El hijo ahora escucha:</p>
  <p>{{ inputText }}</p>
</div>
```

Lanzamos nuestra aplicación con el ng-serve:

We'll use this input to change the name and send it down the component tree

The input component shows the name ⬇️

The name is

¿No es genial? Estamos haciendo que nuestro componente padre hable con su hijo. ¡Tecnología! 🤖

Hola, ¿y cómo podemos hacer que un hijo hable con su padre? Deberían poder comunicarse correctamente, y resolvemos ese problema usando **outputs**.

Outputs

En nuestro **son-component.component.ts** es hora de importar los módulos `Output` y `EventEmitter`:

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

@Component({
  selector: 'app-son-component',
```

```

    templateUrl: './son-component.component.html',
    styleUrls: ['./son-component.component.scss']
  })
  export class SonComponentComponent {
    @Input() inputText: string;

    @Output() emitMessage = new EventEmitter<string>();

    message: string = '';

    sendMessage() {
      this.emitMessage.emit(this.message);
    }
  }
}

```

Vamos a explicar lo que tenemos aquí:

1. **@Output()** inicializa un `EventEmitter` llamado **emitMessage** que devuelve una cadena usando la propiedad **emit**.
2. **message** es una variable de tipo cadena que vamos a cambiar y enviar al componente padre.
3. **sendMessage** es el método que activará la emisión de la variable, utilizando la salida que creamos antes.

El archivo de plantilla **son-component.component.html** se verá así:

```

<!-- Recibe un Input -->
<div>
  <h1>Son Component</h1>
  <p>El hijo ahora escucha:</p>
  <p>{{ inputText }}</p>
</div>
<!-- Emite un Output -->
<div>
  <h1>Envia mensaje al componente padre</h1>
  <!-- ngModel lo iremos aprendiendo poco a poco a lo largo del curso -->
  <input [(ngModel)]="message" />
  <button (click)="sendMessage()">Enviar a padre</button>
</div>

```

¿Qué está pasando aquí?

- La entrada está utilizando un **ngModel** para cambiar dinámicamente el valor de la variable **message**.
- El botón activará el método **sendMessage** al **hacer click**.

¡Y ahora es el momento de recibir los datos!

Basta con adaptar **father-component.component.html**

```

<div>
  <h1>Father Component</h1>
  <input #fatherInput (keyup)="keyUp(fatherInput.value)" />
  <div>
    <!-- Recibe un Input y emite un Output -->
    <app-son-component
      [inputText]="inputText"
      (emitMessage)="setMessage($event)"
    ></app-son-component>
    <h1>Mensaje de mi hijo 📩</h1>
  </div>
</div>

```



```

    <p>{{ sonMessage }}</p>
  </div>
</div>

```

Podemos ver aquí que estamos creando un atributo de evento en **app-son-component** con el mismo nombre de nuestro emisor **emitMessage**. Esto significa que estamos atendiendo el mensaje con **setMessage**, que es un método que vamos a crear ahora en el padre.

Entonces, en **father-component.component.ts** tenemos:

```

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

@Component({
  selector: 'app-father-component',
  templateUrl: './father-component.component.html',
  styleUrls: ['./father-component.component.scss']
})
export class FatherComponentComponent {
  // Se declara e inicia en el componente padre para luego comunicarlo al componente hijo
  inputText: string = '';
  // Variable donde almacenamos el valor del hijo
  sonMessage: string = '';

  // con cada tecla apretada se activa esta funcion.
  keyUp(letra: string) {
    this.inputText = letra;
  }
  // Recibe el mensaje del hijo
  setMessage(message: string): void {
    this.sonMessage = message;
  }
}

```

Antes de nada, para poder hacer uso del **ngModel** y que funcione nuestra aplicación debemos importar en el **app.module** el **FormsModule** que veremos en la próxima lección:

```


import { SonComponentComponent } from './father-component/son-component/son-component.component';
import { FormsModule } from '@angular/forms';

@NgModule({
  declarations: [
    AppComponent,
    FatherComponentComponent,
    SonComponentComponent
  ],
  imports: [
    BrowserModule,
    AppRoutingModule,
    FormsModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }

```

¡Vamos a ver cómo funciona! 🚀

We'll use this input to change the message and output it up to the parent component

And next, we'll see the message 

Una vez conocemos los inputs y outputs debemos conocer una última cosa para ser expertos en comunicación entre componentes. Es el **doblo data binding**.

Banana in a box

Por no extendernos, usamos este operador cuando queremos tanto **enviar** como **recibir** información, para no tener que hacer *input/output* utilizamos el doble data binding o también conocido como banana in a box. Es algo que ya hemos utilizado en nuestro ejemplo anterior con el `ngModel`:

```
<input [(ngModel)]="message" />
```

Ahora que ya has visto directivas, templates, inputs y outputs; es hora de ponerlo a prueba:

<https://gitlab.com/upgrade-hub/ejercicios-master/angular/angular-sesion-2>