

# Angular

Una plataforma open source para desarrollo de aplicaciones web

+Info: <https://angular.io/docs>

# ¿Qué es Angular?

Es una plataforma de código fuente abierto para construcción de aplicaciones clientes escritas HTML combinado con TypeScript.

Está formado por librerías *core* y otras opcionales.

El desarrollo de Angular es liderado por Google sin embargo participan individuos y comunidades de desarrollo.

Es el sucesor de AngularJS o Angular versión 1. Por ello a Angular también se lo conoce como Angular 2+.

Responde al paradigma Single Page Application (SPA)

# ¿Qué es SPA?

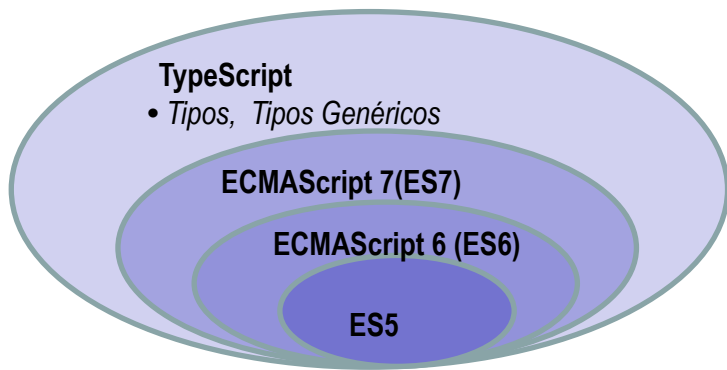
Una aplicación SPA es una aplicación web cuyo objetivo es proveer al usuario una experiencia similar a las aplicaciones de escritorio, con la característica de obtener todo el código HTML, Javascript y CSS a partir de una página. Luego la aplicación podrá seguir descargando progresivamente fragmentos de código para actualizar regiones de dicha página, esto es generalmente en base a interacciones con el usuario.

Algunos frameworks javascript que adoptaron los principios de SPA son [AngularJS](#), [Ember.js](#), [React](#) y [Vue](#)

# ¿Qué es TypeScript?

TypeScript es un lenguaje de programación de código fuente abierto desarrollado y mantenido por Microsoft, con licencia Apache.

Es un superconjunto de JavaScript que esencialmente añade capacidades de POO y objetos basados en clases, la capacidad de chequeo de tipos en compilación (de manera opcional)



TypeScript básicamente incluye JavaScript un poco de C#/Java (sintaxis, herencia, encapsulación, tipos genéricos, ...) y algunos avances propuestos con las últimas versiones de ECMAScript: ES5 , ES6 y ES7.

+info:

<https://www.typescriptlang.org/docs/handbook/intro.html>

TypeScript es un superconjunto de ECMAScript 7, por lo que se **puede mezclar TypeScript con ES5/ES6/ES7 estándar**.

# TypeScript

TypeScript usa el concepto de **transpilación** que **significa compilar el código fuente escrito** en un lenguaje a código fuente de otro lenguaje. En el caso de TypeScript el código se compila a JavaScript.



The screenshot shows a code editor with two tabs: 'TS saludo.ts' and 'JS saludo.js'. The 'TS' tab is active, displaying the following TypeScript code:

```
1 function saludo(person) {  
2   return "Hola, " + person;  
3 }  
4  
5 let user = "Ana Laura";  
6 document.body.innerHTML = saludo(user);
```

The 'JS' tab is also visible, showing the compiled JavaScript code:

```
1 function saludo(person) {  
2   return "Hola, " + person;  
3 }  
4  
5 var user = "Ana Laura";  
6 document.body.innerHTML = saludo(user);
```

**tsc saludo.ts**

saludo.ts



saludo.js

# TypeScript - Tipos de datos

## Variables Tipadas

La principal característica de **TypeScript** por encima de JavaScript es que **permite definir de qué tipo son las variables** que se van a usar.

Supongamos que modificamos la función `saludo(person)`, indicando que el parámetro que se recibe debe ser de tipo `string`



TS saludo.ts	JS saludo.js
1 function saludo(person:string) {	1 function saludo(person) {
2   return "Hola, " + person;	2   return "Hola, " + person;
3 }	3 }
4	4
5 let user = [1,2,3];	5 var user = [1, 2, 3];
6 document.body.innerHTML = saludo(user)	6 document.body.innerHTML = saludo(user);

```
D:\Angular_workspace\Pruebas TypeScript>tsc saludo.ts
saludo.ts:6:34 - error TS2345: Argument of type 'number[]' is not assignable to
parameter of type 'string'.
```

La transpilación indica un error de tipos pero igual finaliza y genera el archivo `saludo.js`.

# TypeScript - Tipos de datos

JavaScript tiene varios tipos de datos diferentes: null, undefined, boolean, number, string, symbol (introducido en ES6), any y de tipo un objeto.

La sintaxis para creación y asignación de variables es:

```
let nombre_var: tipo_dato = valor;
```

Algunos ejemplos:

```
let nombre:String = "Paula";  
let mayor:number = 18;  
let lista1:number[] = [1,2,3,4];  
let lista2:Array<number> = [1,2,3,4];  
let cualquiera1:any = "Paula";  
let cualquiera2:any = 18;
```

En **TypeScript** las variables se declaran con la palabra clave `let`, pero también puede usarse `var`.  
`Let` se usa para variables de alcance local a un bloque de código y `var` para variables globales.

# TypeScript - Clases y objetos

## Clases

TypeScript soporta nuevas características en JavaScript, como el soporte para POO basada en clases. La sintaxis para creación de una clase es la siguiente:

curso.ts  curso.js

```
class Curso {  
  private nombre: string;  
  private horas: number;  
  public constructor(nombre: string,  
                      horas:number){  
    this.nombre = nombre;  
    this.horas = horas;  
  }  
  public setTitulo(nombre: string){  
    this.nombre = nombre;  
  }  
  public getTitulo(){  
    return this.nombre;  
  }  
  public setHoras(horas:number){  
    this.horas=horas;  
  }  
}
```

```
var Curso = /** @class */ (function () {  
  function Curso() {  
  }  
  Curso.prototype.Curso = function (nombre, horas){  
    this.nombre = nombre;  
    this.horas = horas;  
  };  
  Curso.prototype.setTitulo = function (nombre){  
    this.nombre = nombre;  
  };  
  Curso.prototype.getTitulo = function (){  
    return this.nombre;  
  };  
  Curso.prototype.setHoras = function (horas){  
    this.horas = horas;  
  };  
  return Curso;  
})();
```



# TypeScript - Interfaces

Las interfaces comúnmente definen un conjunto de métodos sin implementación, las clases que las implementan deben darle comportamiento. Lo diferente en TypeScript es que una interface puede definir propiedades, mientras que en otros lenguajes las interfaces sólo definen métodos y constantes de clase.

CitaCalendario.ts



CitaCalendario.js

```
interface CitaCalendario {  
  fechaHora: Date;  
  titulo: string;  
  lugar: string;  
}
```

Las interfaces definen  
un nuevo tipo de  
datos.

```
let cita1: CitaCalendario;
```

```
 cita1 = {  
  fechaHora: new Date(Date.now()),  
  titulo: 'Programar en TypeScript',  
  lugar: 'Oficina de Desarrollo'  
}
```

```
var cita1;  
cita1 = {  
  fechaHora: new Date(Date.now()),  
  titulo: 'Programar en TypeScript',  
  lugar: 'Oficina de Desarrollo'  
};
```

Si asignamos un valor que no cumpla con la especificación de la interface, da un error de transpilación.

```
 cita1 = {  
  titulo: 'Programar en TypeScript'  
}
```

```
D:\Angular_workspace\Pruebas>tsc CitaCalendario.ts  
CitaCalendario.ts:9:2 - error TS2322: Type '{ titulo: string; }' is not assignable to type 'CitaCalendario'.  
Property 'fechaHora' is missing in type '{ titulo: string; }'.
```

```
 cita1 = {  
  ~~~~~
```

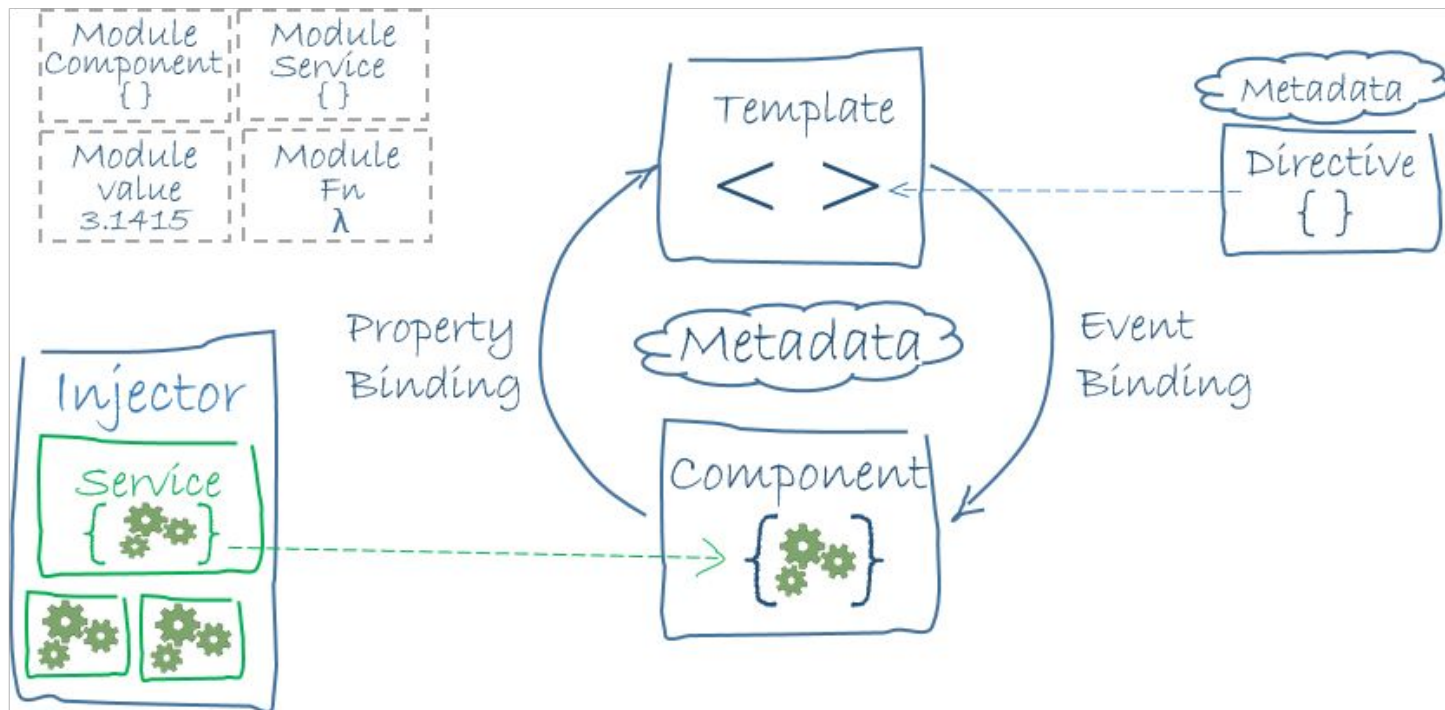
# TypeScript - Decorators

Los *decorators* fueron propuestos en el estándar ECMAScript2016. Los decorators pueden utilizarse sobre clases, métodos, propiedades y parámetros.

Angular define varios decorators que agregan metadatos a las clases, para indicar al sistema el significado de esas clases y cómo esas clases deberían trabajar.

+info: <https://www.typescriptlang.org/docs/handbook/decorators.html>

# La arquitectura de Angular



Extraído de <https://angular.io/guide/architecture>

# ¿Cómo se escriben aplicaciones en Angular?

- componiendo *templates* HTML con tags *Angularizados*,
- escribiendo *componentes* en TypeScript que gerencian estas plantillas,
- incorporando lógica de aplicación a través de *servicios*, en TypeScript y,
- empaquetando componentes y servicios en módulos.

Las *componentes* y los *templates* interactúan constituyendo la vista de la aplicación. Las componentes le proveen los datos a los templates.

Los *metadatos* le indican a Angular cómo procesar las clases, por ejemplo se podrían utilizar para relacionar los *componentes* con los *templates*. Los metadatos se indican mediante *decorators* de TypeScript.

Las *directivas* actúan sobre la vista, modificando el DOM generado por el *template*.

# Angular en ejemplos

*hero-list.component.ts*

Decorator **@Component** transforma una clase en un **Componente**

```
@Component({  
  selector: 'hero-list',  
  templateUrl: './hero-list.component.html'  
})
```

**Metadatos**

**Template**

```
export class HeroListComponent implements OnInit {  
  /* ... */  
}
```

**Clase HeroListComponent**

*hero-list.component.html*

```
<h2>Hero List</h2>  
<p><i>Pick a hero from the list</i></p>  
<ul>  
  <li *ngFor="let hero of heroes" (click)="selectHero(hero)">  
    {{hero.name}}  
  </li>  
</ul>  
<hero-detail *ngIf="selectedHero" [hero]="selectedHero"></hero-detail>
```

# Módulos

Un **módulo** es una clase que describe cómo se unen las diferentes partes de la aplicación.

Todas las aplicaciones Angular tienen al menos un **módulo**: el módulo root que arranca la aplicación, usualmente llamado **AppModule**.

Las apps Angular son modulares y el mismo framework provee diferentes módulos para construirlas.

En Angular los módulos se definen mediante el decorador **@NgModule**.

```
@NgModule({  
  imports:    [ BrowserModule ],  
  declarations: [ AppComponent ],  
  bootstrap:  [ AppComponent ] ← Despliega la primera vista  
})  
export class AppModule { }
```

# Mi primer App: Hola Angular



## *src/app/app.module.ts*

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

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

## *src/app/app.component.ts*

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
  template: `<h1>Hola {{name}}</h1>`
})
export class AppComponent { name = 'Angular';
}
```

## *src/main.ts*

```
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
import { AppModule } from './app/app.module';

platformBrowserDynamic().bootstrapModule(AppModule);
```

# Inicio de una aplicación Angular

Existen muchas maneras de iniciar una aplicación Angular, dependiendo de cómo se desea compilarla o dónde ejecutarla.

Durante el **desarrollo**, generalmente se compila la aplicación dinámicamente con un compilador JIT (Just-In-Time) y se ejecuta en un navegador.

La forma de inicio (bootstrap) se especifica en el archivo **src/main.ts**

El módulo root (**AppModule**) crea una instancia de **AppComponent** (especificado en el atributo bootstrap del objeto metadata) y lo inserta dentro del tag identificado por su selector (**app-root**).

Angular busca el tag **<app-root>** en el archivo **index.html** y despliega ahí el componente.



# Inicio de una aplicación Angular

El archivo **index.html** es la página principal de la aplicación. Generalmente no es necesario modificarla durante el desarrollo. Al usar **Angular-CLI**, los archivos *js* y *css* son agregados automáticamente durante el proceso de “*building*” de la aplicación.

*src/index.html*

```
<app-root><!-- contenido manejado por Angular --></app-root>
```

*src/app/app.component.ts*

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

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



# Uso de templates y binding de datos

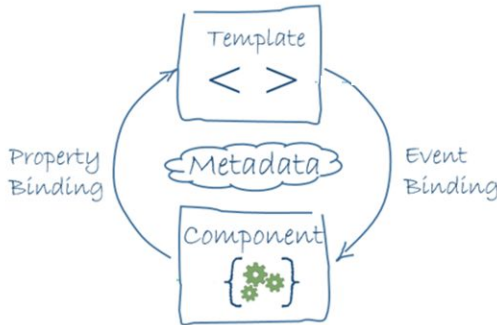
# Angular - Data binding

El *data binding* en Angular es la comunicación o el enlace de datos entre el archivo Typescript del componente y el archivo HTML de su *Template*. Angular provee varias formas de llevar a cabo los procesos *data binding*.

*Binding one way de la fuente de datos a la vista:* *interpolación* o *property binding*.

*Binding one way de la vista a la fuente de datos:* *event binding*.

*Binding two-way*, es decir, enlazar en ambos sentidos y simultáneamente los datos, de la vista a la fuente de datos y viceversa.



# Data binding one way: Interpolación

La forma más fácil es usando **interpolación**. Con interpolación es posible usar el nombre de una propiedad de la componente –o una expresión angular- en el *template*, encerrándola entre doble llaves.

app.component.ts

```
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
})
export class AppComponent {
  nombre = 'CURSO ANGULAR';
  horario = '16 a 20 Hs.';
}
```

app.component.html

```
<div style="text-align:center">
  <h1>
    Bienvenidos al curso {{nombre}}!
    Horario: {{horario}}
  </h1>
</div>
```

app.component.ts

```
@Component({
  selector: 'app-root',
  template: `<h1>Bienvenido al {{nombre}}</h1>
    <h2>Horario: {{horario}}</h2>
  `
})
export class AppComponent {
  nombre = 'CURSO ANGULAR';
  horario = '16 a 20 Hs.';
}
```

*El template es un string multilinea, ECMAScript 2015 permitió el uso de backticks: (``) para denotarlo. No es lo mismo que comilla simple: (')*

# Data binding one way: Interpolación

También se puede usar interpolación con propiedades de un objeto del componente como origen de los datos.

Una de las formas para crear objetos en Angular es mediante un modelo, es decir, podemos crear un archivo en el que se declara la clase y sus propiedades para posteriormente poder utilizarla en los componentes, de ahí que denomine a estos archivos como modelos.

alumno.modelo.ts

```
export class Alumno {
  public id: number;
  public nombre: string; ①
  public apellidos: string;
  public ciudad: string;

  constructor (id: number, nombre: string,
    apellidos: string, ciudad: string){
    this.id = id;
    this.nombre = nombre;
    this.apellidos = apellidos; ②
    this.ciudad = ciudad;
  }
}
```

app.component.html

```
<div>
  <h4>Información del Alumno</h4> <hr>
  <h5>id: {{ alumno1.id }}</h5> <hr>
  <h5>Nombre: {{ alumno1.nombre }}</h5> <hr>
  <h5>Apellidos: {{ alumno1.apellidos }}</h5><hr>
  <h5>Ciudad: {{ alumno1.ciudad }}</h5> <hr>
</div>
```

 *nombre objeto.propiedad objeto*

app.component.ts

```
import { Alumno } from '../modelos/alumno.modelo';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
})
export class AppComponent {
  alumno1 = new Alumno(1, 'Ana', 'Rios', 'La Plata');
}
```

# Data binding one way: Property binding

Property Binding es el segundo tipo de data binding empleado por Angular para enlazar valores de la fuente de datos a la vista. En este caso, se trata de un enlace que relaciona un atributo con una expresión, con la siguiente sintaxis:

**[atributodelelementoHTML] = “expresión”**

**ejpropertybinding.component.ts**

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

@Component({
  selector: 'app-ejpropertybinding',
  templateUrl: './ejpropertybinding.component.html',
})
export class EjpropertybindingComponent implements OnInit {

  constructor() {
    texto = 'Escribe algo';
  }

  ngOnInit() {
    setTimeout(() => {
      this.texto = 'por favor';
    }, 3000);
  }
}
```

*cambia el valor 'Escribe algo...'  
de la propiedad texto a 'por favor'  
cuando pasen 3 segundos.*

**ejpropertybinding.component.html**

```
<p>
<input type="text" placeholder="Escribe algo . . .">
<input type="text" [placeholder]="texto">
</p>
```

Property binding

**Ejemplo de Property Binding**

Escribe algo...

por favor

# Data binding one way: Event binding

Los enlaces *event binding* son enlaces de un solo sentido, *one way*, pero en este caso desde la vista a la fuente de datos, ya que como su nombre indica, los desencadena un evento en el cliente web.

En este enlace de datos, se iguala un evento de un elemento HTML de la vista con una expresión, normalmente un método definido en la clase del componente con la sintaxis:

**evento="nombreMetodo()";**

## ejeventbinding.component.ts

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

@Component({
  selector: 'app-ejeventbinding',
  templateUrl: './ejeventbinding.component.html',
})
export class EjeventbindingComponent implements OnInit {
  texto = 'Originalmente el texto se carga así';

  modTexto() {
    this.texto = 'Al pulsar el botón el texto se muestra así';
  }
  ngOnInit() { }
}
```



Al pulsar el botón el texto se muestra así

## ejeventbinding.component.html

```
<button class="btn btn-success"
  (click)="modTexto()">
  Modificar Texto</button>
<h3> {{ texto }} </h3>
<hr>
```

Event binding

# Two way binding

El último tipo de enlace es *two way binding*, que enlazan en ambos sentidos y simultáneamente, los datos de la vista a la fuente de datos y viceversa.

Este enlace emplea una mezcla entre la sintaxis de eventos y la de interpolación, donde se *enlaza el valor del elemento HTML de la vista* con la propiedad del componente, de la siguiente manera:

**`[(directiva)] = "nombredelapropiedad"`**

## ej2waybinding.component.ts

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

@Component({
  selector: 'app-ej2waybinding',
  templateUrl: './ej2waybinding.component.html',
  styleUrls: ['./ej2waybinding.component.css']
})
export class Ej2waybindingComponent implements OnInit {
  texto = 'Texto original al cargar';

  constructor() { }
  ngOnInit() { }
}
```

## ej2waybinding.component.html

```
<label>Ingrese un valor:</label>
<input type="text" class="form-control"
  [(ngModel)]="texto">
<h3>{{texto}}</h3>
```

Two way binding

En este caso hemos utilizado la directiva `ngModel` que más adelante detallaremos en el apartado correspondiente.



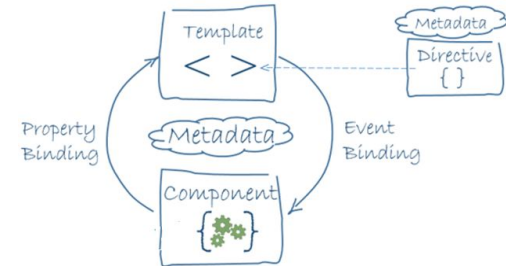
# Directivas

Las directivas son clases Angular con instrucciones para crear, formatear e interactuar con los elementos HTML en el DOM, y son una de las principales características de este *framework*. Se declaran mediante el empleo del decorador **@Directive** y pueden ser agregadas a cualquier tag en el template.

Angular incluye un gran número de directivas con funcionalidades típicas que se necesitan en una aplicación.

Existen tres tipos de directivas:

- **Componentes**. Están relacionadas con las etiquetas HTML de cada componente, que renderizan su *template* donde las ubiquemos. Se podría decir que el decorador `@Component` es una aplicación de las directivas.
- **De atributos**. Son directivas que modifican el comportamiento de un elemento HTML de la misma manera que un atributo HTML, es decir, sin modificar el layout o presentación del elemento en el DOM. Se identifican por tener el prefijo `ng`.
- **Estructurales**. Estas directivas alteran el layout del elemento en el que se asocian, añadiendo, eliminando o reemplazando elementos en el DOM. Se identifican sobre las directivas de atributo por ir precedidas del símbolo asterisco (\*).



# La directiva \*ngIf

**\*ngIf** es una directiva estructural de Angular para implementar estructuras if en nuestra aplicación. Con esta directiva condicionamos que un elemento html se muestre o no en función de que se cumpla la expresión que define, normalmente una propiedad o método. La sintaxis básica es:

**\*ngIf="expresión/propiedad/metodo"**

## ejdirectivangif.component.ts

```
import { Component, OnInit } from '@angular/core';
@Component({
  selector: 'app-ejdirectivangif',
  templateUrl: './ejdirectivangif.component.html',
})
export class EjdirectivangifComponent implements OnInit {
  nombre: string;
  constructor() { }
  ngOnInit() {
  }
}
```



*El botón estará oculto hasta que se comience a escribir ya que la directiva localiza la existencia de la propiedad asociada.y si tiene contenido mostrará el botón.*

## ejdirectivangif.component.html

```
<div class="container">
  <label>Nombre y Apellidos: </label>
  <input type="text" class="form-control"
    [(ngModel)]="nombre" placeholder="Complete su nombre y apellidos">
  <button type="submit" class="btn btn-primary"
    *ngIf="nombre">Enviar</button>
</div>
```

# La directiva \*ngFor

Una de las directivas más empleadas en las aplicaciones Angular es la ngFor, dedicada a realizar iteraciones y presentar listados por pantalla. Se trata de una directiva de tipo estructural y su sintaxis es:

`*ngFor="let objeto/propiedad of objetos/propiedades"`

`ejdirectivangfor.component.ts`

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

@Component({
  selector: 'app-ejdirectivangfor',
  templateUrl: './ejdirectivangfor.component.html',
})
export class EjdirectivangforComponent implements OnInit {
  cursos: string[];
  constructor() {
    this.cursos = ['Angular', 'Html', 'CSS'];
  }
  ngOnInit() {
  }
}
```

Cursos Disponibles

- Angular
- HTML
- CSS

`ejdirectivangfor.component.html`

```
<h3>Cursos Disponibles</h3>
<ul>
  <li *ngFor="let curso of cursos">
    <h4>{{curso}}</h4>
  </li>
</ul>
```

Se crea dentro del elemento html con una variable local (curso) con **let** que recorrerá el array proveniente del componente definido por **of** (cursos).

Como buena práctica, el nombre de la variable local será singular y el del array plural.

Declaración de variable que contiene el valor de la iteración      Variable a iterar      Índice de la iteración

```
<div *ngFor="let elemento of miArray; let i = index">
  <p>El valor de párrafo es {{elemento}}</p>
</div>
```

Valor de la iteración

# Angular - \*ngIf y \*ngFor desde versión 17

## Angular 17

```
<div *ngIf="showTable; else showList">
  <table>
    <!-- full table -->
  </table>
</div>

<ng-template #showList>
  <ul>
    <li><!-- full list --></li>
  </ul>
</ng-template>

<div *ngFor="let item of items; let i = index;">
  <p>{{ item }}</p>
</div>
```

```
<!-- *ngIf -->
@ngIf (showTable) {
  <table>
    <!-- full table -->
  </table>
} @else {
  <ul>
    <li><!-- full list --></li>
  </ul>
}

@for (item of items; track item.id) {
  <p>{{ item }}</p>
}
```

# Atributos HTML vs propiedades DOM

- Los **atributos HTML** se utilizan para *inicializar* propiedades DOM.
- Los valores de las propiedades pueden cambiar, los valores de los atributos no.
- Por ejemplo, el navegador al renderizar `<input type="text" value="Bob">` crea un nodo DOM con el valor de la propiedad inicializada en *"Bob"*
- Cuando el usuario escribe "Sally" en el input, **la propiedad del DOM se actualiza** pero el valor del atributo HTML no cambia.
- Los atributos HTML y las propiedades DOM aunque tengan el mismo nombre son cosas distintas.

# Template expressions

Una expresión **produce un valor**. Por ej: `{{1+1}}`

Deberían seguir las siguientes reglas: sin efectos colaterales visibles, ejecución rápida, simple, idempotente.

Angular ejecuta la expresión y **la asigna a una propiedad de elemento HTML o un componente o una directiva**.

Se escriben en lenguaje Javascript, aunque algunas expresiones que involucren efectos colaterales no se permiten.

- asignaciones (=, +=, -=, ...)
- new
- expresiones encadenadas con ; o ,
- operadores de incremento y decremento (++ y --)

Se incluyen nuevos operadores: | (operador pipe), ?. y !. (ambos para tratar valores nulos)

```
<div>Birthdate: {{currentHero?.birthdate | date:"MM/dd/yy"}}</div>
```

Angular viene con algunos Pipes built-in: **date**, **uppercase**, **lowercase**, **currency**, y **percent**

# Template statements

**Responden a un evento** disparado por un elemento, o un componente o una directiva.

```
<button (click)="deleteHero()">Delete hero</button>
```

```
<button *ngFor="let hero of heroes" (click)="deleteHero(hero)">{{hero.name}}</button>
```

Normalmente tienen efectos colaterales, ya que **sirven para actualizar la aplicación a partir de las acciones del usuario**.

Se escriben en lenguaje Javascript, soportando asignaciones y expresiones encadenadas, pero no son permitidas las siguientes:

- new
- operadores de incremento y decremento ++ y --
- operadores de asignación, como += y -=
- | (operador pipe), ?. y !.

# Binding, expressions y statements

Pueden ser agrupados en tres categorías según la dirección del flujo de datos:

- source-to-view
- view-to-source
- view-to-source-to-view

Flujo de datos	Sintaxis	Tipo
Sentido único. De fuente de datos a vista	<code>{{<b>expression</b>}}</code> <code>[target]="<b>expression</b>"</code> <code>bind-target="<b>expression</b>"</code>	Interpolation Property Attribute Class Style
Sentido único. De vista a fuente de datos	<code>(target)="statement"</code> <code>on-target="statement"</code>	Event
Doble sentido	<code>[(target)]="<b>expression</b>"</code> <code>bindon-target="<b>expression</b>"</code>	Doble sentido



# Template expression vs String literal

Los corchetes le indican a Angular que debe evaluar la expresión.

Sin los corchetes en este caso trata de asignar un string a una propiedad que es de tipo Hero.

```
<!-- ERROR: HeroDetailComponent.hero expects a  
      Hero object, not the string "currentHero" -->  
<app-hero-detail  
  hero="currentHero"></app-hero-detail>
```

```

```

*asigna el string: heroImageUrl, debería ser directamente la url -> src="http://www.dominio.com/imagen.jpg"*

```
<img [src]="heroImageUrl">
```

*evalúa el template expression y hace el property binding (heroImageUrl puede ser una propiedad de un componente)*

# Ejemplo

Dependiendo del tipo, el target puede ser una propiedad o un evento de un elemento, componente, directiva; o raramente un nombre de atributo:

```
<h1>
  <!--interpolación de variables definidas en el modelo del componente-->
  {{title}}
</h1>
<form>
  <label>¿Cómo te llamas?</label>
  <!--enlace doble (lectura y escritura) entre la vista y el modelo-->
  <input type="text" [(ngModel)]="aprendiz" />
  <!--interpolación-->
  <p>Bienvenido a Angular {{ aprendiz }} </p>

  <!--template expression-->
  <p>Soy capaz de multiplicar por {{1 * 2}} tus habilidades </p>
  <!--asignación de un template statement a un evento-->
  <button (click)="hacerVisible()">Saludar</button>
  <!--asociación de un template expression a una propiedad-->
  <p [hidden]="visible">Hola Mundo!!!</p>
</form>
```

↑  
*Una template expression que hace un property binding, la propiedad del componente se llama visible*

# Tips de instalación y ejecución

## Pre-requisitos:

Node.js version **current**, **active LTS** o **maintenance LTS**

-> <https://nodejs.org/es/download/>

Para chequear la versión instalada se puede usar el comando -> `node -v`

NodeJS es un entorno de ejecución para JavaScript.

npm es gestor de paquetes de nodeJS.

Angular, Angular CLI, y las aplicaciones Angular dependen de librerías que están disponibles en paquetes npm.

Para descargar los paquetes npm se debe usar un npm package manager.

Node.js instala por defecto un cliente npm de línea de comandos.

Para chequear el cliente se puede ejecutar `npm -v`

# Tips de instalación y ejecución

Una vez instalado el cliente npm se puede usar para instalar Typescript:

```
> npm install -g typescript
```

Chequear la instalación:

```
> tsc --version
```

```
> tsc --help
```

Angular CLI es una herramienta que ayuda a configurar un proyecto Angular desde la línea de comandos. Para instalarla de manera global:

```
> npm install -g @angular/cli (+info: https://cli.angular.io/)
```

Creación de un proyecto

```
> ng new nomProyecto
```

Creación de una componente

```
> ng generate component --spec false nomComponente
```

Para probar la aplicación arrancamos un servidor de Angular desde la carpeta del proyecto:

```
> ng serve
```

Si termina bien acceder a **<http://localhost:4200>**

# Uso de formularios

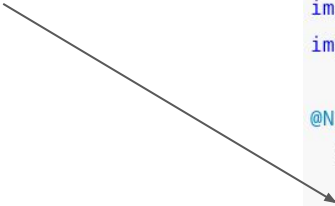
# Template reference variables ( #var )

Generalmente se utiliza para referenciar a elementos del DOM dentro de un template. Se utiliza el símbolo # para declararla.

```
<input #phone placeholder="phone number">  
  
<!-- lots of other elements -->  
  
<!-- phone refers to the input element; pass its `value` to an event  
handler -->  
  
<button (click)="callPhone(phone.value)">Call</button>
```

# Forms

Para utilizar forms es necesario importar **FormsModule** en el arreglo de imports del modulo de la aplicación



*src/app/app.module.ts*

```
import { NgModule }      from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { FormsModule }    from '@angular/forms';

import { AppComponent }  from './app.component';
import { HeroFormComponent } from './hero-form/hero-form.component';

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

# Forms

## Formulario para alta de héroes

src/app/hero-form/hero.ts

```
export class Hero {  
  constructor(  
    public id: number,  
    public name: string,  
    public power: string,  
    public alterEgo?: string  
  ) { }  
}
```

*El atributo alterEgo es opcional,  
así que el constructor permite que  
se omita*

src/app/hero-form/app.component.html

```
<app-hero-form></app-hero-form>
```

src/app/hero-form/hero-form.component.ts

```
import { Component } from '@angular/core';  
import { Hero } from '../hero';  
import { FormsModule } from '@angular/forms';  
  
@Component({  
  selector: 'app-hero-form',  
  templateUrl: './hero-form.component.html',  
  styleUrls: ['./hero-form.component.css']  
})  
export class HeroFormComponent {  
  
  powers = ['Really Smart', 'Super Flexible',  
            'Super Hot', 'Weather Changer'];  
  model = new Hero(18, 'Dr IQ', this.powers[0], 'Chuck Overstreet');  
  submitted = false;  
  
  onSubmit() { this.submitted = true; }  
  
  // TODO: Remove this when we're done  
  get diagnostic() { return JSON.stringify(this.model); }  
}
```



# Forms

src/app/hero-form/hero-form.component.html

```
<form (ngSubmit)="onSubmit()" #heroForm="ngForm">

  {{diagnostic}}

  <div class="form-group">
    <label for="name">Name</label>
    <input type="text" class="form-control" id="name"
      required
      [(ngModel)]="model.name" name="name">
  </div>

  <div class="form-group">
    <label for="alterEgo">Alter Ego</label>
    <input type="text" class="form-control" id="alterEgo"
      [(ngModel)]="model.alterEgo" name="alterEgo">
  </div>

  <div class="form-group">
    <label for="power">Hero Power</label>
    <select class="form-control" id="power"
      required
      [(ngModel)]="model.power" name="power">
      <option *ngFor="let pow of powers" [value]="pow">{{pow}}</option>
    </select>
  </div>

  <button type="submit" class="btn btn-success"
    [disabled]="!heroForm.form.valid">Submit</button>
```

Solo para prueba luego se borra

Utiliza  
validación de  
HTML5

## Hero Form

{ "id": 18, "name": "Dr IQ 3000", "power": "Super Flexible", "alterEgo": "Chuck OverUnderStreet" }

Name

Alter Ego

Hero Power

La variable `#heroForm` en este caso es una referencia a una directiva de Angular llamada `NgForm`. `NgForm` tiene la capacidad de hacer un seguimiento de los valores y validez de cada control del formulario.

# Forms

También es posible usar **NgForm** de esta manera

src/app/hero-form/hero-form.component.html

```
<form #heroForm="ngForm" (ngSubmit)="onSubmit(heroForm)">

<div class="form-group">
  <label for="name">Name</label>
  <input ngModel type="text" class="form-control" id="name"
    required name="name">
</div>
...

<button type="submit" class="btn btn-success"
[disabled]="!heroForm.valid">Submit</button>
```

El atributo name se utiliza si el input se encuentra dentro de un form. Establece el nombre clave para acceder desde el NgForm.

clave: name

src/app/hero-form/hero-form.component.ts

```
import { _Component } from '@angular/core';
import { Hero } from '../hero';
import { _FormsModule } from '@angular/forms';

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

  powers = ['Really Smart', 'Super Flexible',
    'Super Hot', 'Weather Changer'];

  model = new Hero(18, 'Dr IQ', this.powers[0], 'Chuck Overstreet');

  submitted = false;

  onSubmit(formulario: NgForm) {
    if(formulario.valid) {
      this.model.name = formulario.value.name;
      ...
      this.submitted = true;
    }
  }

  // TODO: Remove this when we're done
  get diagnostic() { return JSON.stringify(this.model); }
}
```

# Forms

**NgModel** lleva un seguimiento de cada control. Informa si el usuario tocó el control, si el valor cambió o si el valor es inválido.

**NgModel** actualiza el control estableciendo el atributo *class* con valores CSS propios de Angular



State	Class if true	Class if false
The control has been visited.	ng-touched	ng-untouched
The control's value has changed.	ng-dirty	ng-pristine
The control's value is valid.	ng-valid	ng-invalid

Para testear los valores se puede hacer lo siguiente:

```
....  
<input type="text" class="form-control" id="name"  
  required  
  [(ngModel)]="model.name" name="name" #spy>  
  
<br>TODO: remove this: {{spy.className}}  
....
```

src/app/hero-form/hero-form.component.htm

```
....  
<label for="name">Name</label>  
<input type="text" class="form-control" id="name"  
  required  
  [(ngModel)]="model.name" name="name"  
  #name="ngModel">  
  
  <div [hidden]="name.valid || name.pristine"  
    class="alert alert-danger">  
    Name is required  
  </div>  
....
```

# Uso de routing

# Routing

El Routing nos permite navegar desde una vista hacia otra, de acuerdo a las acciones del usuario.

- Utiliza el modelo de navegación del navegador de internet
- Interpreta la **URL** del navegador como una instrucción para navegar **hacia una vista**.
- **Es posible pasar parámetros** al componente de la vista para tomar decisiones sobre qué contenido mostrar.
- El routing es configurado con mapeos entre links y vistas, denominados **rutas**.
- Es posible navegar cuando se cliquea un enlace, se cliquea un botón, se selecciona un elemento de un dropdown, etc.
- **Registra la navegación en el historial del navegador**, permitiendo el uso de los botones Adelante/Atrás.
- No es parte de Angular core, se encuentra en su propio package **@angular/router**

```
import { RouterModule, Routes } from '@angular/router';
```

# Routing - Configuración de rutas

Un ruta básicamente tiene dos propiedades:

- **path**: un string que contiene la URL que se usa en la barra de direcciones del navegador
- **component**: el componente que el ruteador creará cuando se navega a esta ruta

```
import { HeroesComponent }    from './heroes/heroes.component';

const routes: Routes = [
  { path: 'heroes', component: HeroesComponent }
];
```

*Con la dirección localhost:4200/heroes navegaremos al componente HeroesComponent*

# Routing - Configuración de rutas

Primeramente agregar `<base href="/">` dentro del tag `<head>` de index.html, para indicar cómo se componen las URLs.

Luego se define un arreglo de rutas utilizando la interface `Route` (type `Routes = Route[];`)

*src/app/app.module.ts*

```
import { NgModule }           from '@angular/core';
import { BrowserModule }      from '@angular/platform-browser';
import { FormsModule }        from '@angular/forms';
import { RouterModule, Routes } from '@angular/router';

import { AppComponent }       from './app.component';
import { ComponentOne }       from './component-one';
import { ComponentTwo }       from './component-two';

const appRoutes: Routes = [
  { path: 'component-one', component: ComponentOne },
  { path: 'component-two', component: ComponentTwo }
];
```

```
@NgModule({
  imports: [
    BrowserModule,
    FormsModule,
    RouterModule.forRoot(
      appRoutes,
      { enableTracing: true } //<--debugging purposes only
    )
  ],
  declarations: [
    AppComponent,
    ComponentOne,
    ComponentTwo,
  ],
  bootstrap: [ AppComponent ]
})
export class AppModule { }
```

*RouterModule.forRoot* toma el arreglo de rutas y retorna un módulo router configurado.

# Routing - Configuración de rutas

interface Route

```
interface Route {  
  path?: string  
  pathMatch?: string  
  matcher?: UrlMatcher  
  component?: Type<any>  
  redirectTo?: string  
  outlet?: string  
  canActivate?: any[]  
  canActivateChild?: any[]  
  canDeactivate?: any[]  
  canLoad?: any[]  
  data?: Data  
  resolve?: ResolveData  
  children?: Routes  
  loadChildren?: LoadChildren  
  runGuardsAndResolvers?: RunGuardsAndResolvers  
}
```

*app/component-one.ts*

```
import {Component} from '@angular/core';  
  
@Component({  
  selector: 'component-one',  
  template: 'Component One'  
})  
export default class ComponentOne {  
}
```

*app/component-two.ts*

```
import {Component} from '@angular/core';  
  
@Component({  
  selector: 'component-two',  
  template: 'Component Two'  
})  
export default class ComponentTwo {  
}
```



# Routing - Redireccionamiento de rutas

Al comenzar la aplicación navega a la ruta vacía. Podemos redireccionar la ruta vacía a una nombrada:

*app/app.routes.ts*

```
...  
const appRoutes: Routes = [  
  → { path: '', redirectTo: 'component-one', pathMatch: 'full' },  
    { path: 'component-one', component: ComponentOne },  
    { path: 'component-two', component: ComponentTwo }  
];
```

*pathMatch: indica si toda la URL debe hacer match o solo la última parte (por ej, para el caso de rutas hijas)*

# Routing - Navegación

Se suele utilizar el wildcard \*\* como última ruta.

```
{ path: '**', component: PageNotFoundComponent }
```

El router selecciona esta ruta si la url no hace match con ningún path. Es muy usado para mostrar "404 - Not Found" o redireccionar a otra ruta.

El **orden de la rutas** en la configuración **es importante**.

El router usa la estrategia *first-match wins*, de esta manera las rutas **más específicas** deberán ubicarse **arriba** de las **menos específicas**.

# Routing - Navegación

Usando la directiva **RouterLink**, por ejemplo para definir un link hacia component-one

Para links estáticos

```
<a routerLink="/component-one">Component One</a>
```

Para links con valores dinámicos

```
<a [routerLink]="['/component-one']">Component One</a>
```

Se le pasa un arreglo de segmentos path

Por ejemplo: `['/team', teamId, 'user', userName]` --> `/team/11/user/bob`

Navegando a una ruta programáticamente llamando a la función ***navigate*** del router

```
this.router.navigate(['/component-one']);
```

# Routing - RouterOutlet

**RouterOutlet** funciona como un contenedor para mostrar componentes.

Angular ubica el componente en el elemento: `<router-outlet></router-outlet>`

*app/app.component.ts*

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

@Component({
  selector: 'app',
  template: `
    <nav>
      <a routerLink="/component-one">Component One</a>
      <a routerLink="/component-two">Component Two</a>
    </nav>
    <div style="color: green; margin-top: 1rem;">Outlet:</div>
    <div style="border: 2px solid green; padding: 1rem;">

      <router-outlet></router-outlet>
      <!-- Route components are added by router here -->
    </div>
  `
})
export class AppComponent {}
```

*app/component-one.ts*

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

@Component({
  selector: 'component-one',
  template: 'Component One'
})
export default class ComponentOne {
}
```

Component One Component Two

Outlet:

Component One

# Routing - RouterOutlet auxiliar

Es posible definir contenedores auxiliares asignándoles nombres únicos, de la siguiente manera:

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

@Component({
  selector: 'app',
  template: `
    <nav>
      <a [routerLink]="['/component-one']">Component One</a>
      <a [routerLink]="['/component-two']">Component Two</a>
      <a [routerLink]="[{ outlets: { 'sidebar': ['component-aux'] } }]">Component Aux</a>
    </nav>

    <div style="color: green; margin-top: 1rem;">Outlet:</div>
    <div style="border: 2px solid green; padding: 1rem;">
      <router-outlet></router-outlet>
    </div>

    <div style="color: green; margin-top: 1rem;">Sidebar Outlet:</div>
    <div style="border: 2px solid blue; padding: 1rem;">
      <router-outlet name="sidebar"></router-outlet>
    </div>
  `
})
export class AppComponent {
}
```

```
export const routes: Routes = [
  { path: '', redirectTo: 'component-one', pathMatch: 'full' },
  { path: 'component-one', component: ComponentOne },
  { path: 'component-two', component: ComponentTwo },
  { path: 'component-aux', component: ComponentAux, outlet: 'sidebar' }
];
```

# Routing - RouterLinkActive

Esta directiva agrega o elimina atributos class en un elemento HTML cuando un *routerLink* se encuentra activo o inactivo respectivamente.

```
<h1>Angular Router</h1>
<nav>
  <a routerLink="/crisis-center" routerLinkActive="active">Crisis Center</a>
  <a routerLink="/heroes" routerLinkActive="active">Heroes</a>
</nav>
<router-outlet></router-outlet>
```

Utilizado por ejemplo para agregar el atributo **active** de bootstrap automáticamente en el link activo



```
<nav class="navbar navbar-default">
  <div class="container-fluid">
    <div class="navbar-header">
      <a class="navbar-brand" href="#">WebSiteName</a>
    </div>
    <ul class="nav navbar-nav">
      <li class="active"><a href="#">Home</a></li>
      <li><a href="#">Page 1</a></li>
      <li><a href="#">Page 2</a></li>
      <li><a href="#">Page 3</a></li>
    </ul>
  </div>
</nav>
...
```

