



Capitolo 1 – Introduzione

Cosa è Angular?

Angular è un UI framework creato da Google strutturato in componenti e di uso comune per la creazione di SPA Enterprise level.

È strutturabile in **componenti**, mappabili in un albero, costituito da componenti padre e componenti figlio.

Componente Padre

|

|

+ --- Componente Figlio

Perché usare Angular?

- Design Pattern ben strutturato
- Usa TypeScript
- Molti strumenti utili
- Componenti disaccoppiati (I componenti possono eseguire tasks separatamente)
- Il markup è chiaro e semplice, permettendo una manipolazione DOM agile
- Testing friendly
- Grande community di supporto
- Molteplici piattaforme d'uso

Struttura di un'applicazione in Angular

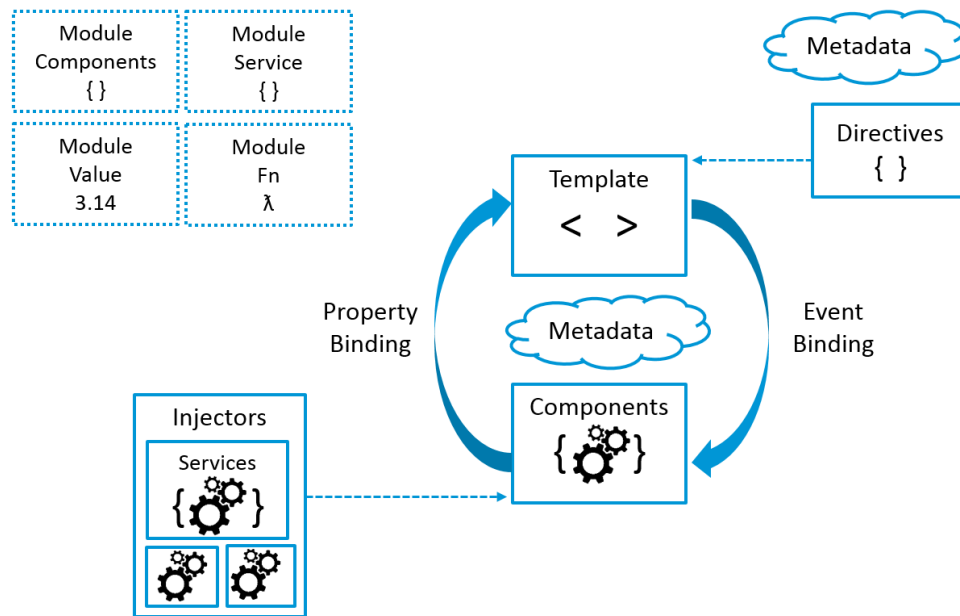


Figura 1 Schema di una SPA tipo in Angular

1) Moduli Angular (o NG Modules)

Ogni app Angular ha **almeno una** Angular Module Class, la **root module** (o **app module**)

2) Template

I Templates di Angular sono **dinamici**, quando le **directives** danno le istruzioni, il DOM verrà manipolato per visualizzare la pagina in un modo.

3) Metadati

Dicono ad Angular come processare una classe

4) Service

Un'ampia categoria che raccoglie tutte quelle funzioni di cui l'applicazione ha bisogno.

I componenti ne fanno largo uso.

Es.: Può essere, ad esempio, una classe con uno scopo ben specifico.

5) Componenti

Blocco essenziale di un'applicazione Angular

Un Componente è costituito da:

|

- **HTML Template** (dichiara cosa verrà renderizzato)
- Una **Classe in TypeScript** (definisce cosa deve fare)
- **CSS Selector** (definisce come un componente è usato nel template)
- **CSS Style** (danno lo stile)

Un componente deve appartenere ad un **NgModule**, affinché possa interagire con **un altro componente**.

Capitolo 2 – Installare un ambiente Angular

- Installare Angular CLI
- Creare un Angular Workspace

- 1) `npm install -g @angular/cli`
- 2) `ng new my-app`

Capitolo 3 – Creare un nuovo Componente

My-app>app>nome_componente.component.ts

```
import { Component } from "@angular/core";  
// Import delle features di Angular
```

```
@Component({  
  selector: 'app-hello-world',  
  template: '<h1>{{title}}</h1>',  
  styles: [`  
    h1 {  
      color: blue;  
    }`]  
})  
  
export class HelloWorldComponent {  
  title = 'Hello World!';  
}
```

- 1) Aggiungere un **Decorator**

```
@Component({})
```

- 2) Un **decorator** marca una classe in un componente. Questo decorator ha dei metadati che definiscono come il componente si dovrà comportare in runtime.

```
selector: 'app-hello-world',
```

- 3) Inseriamo un CSS Selector per marcare il nostro componente che verrà usato nel nostro template file per renderizzare il componente

```
template: '',
```

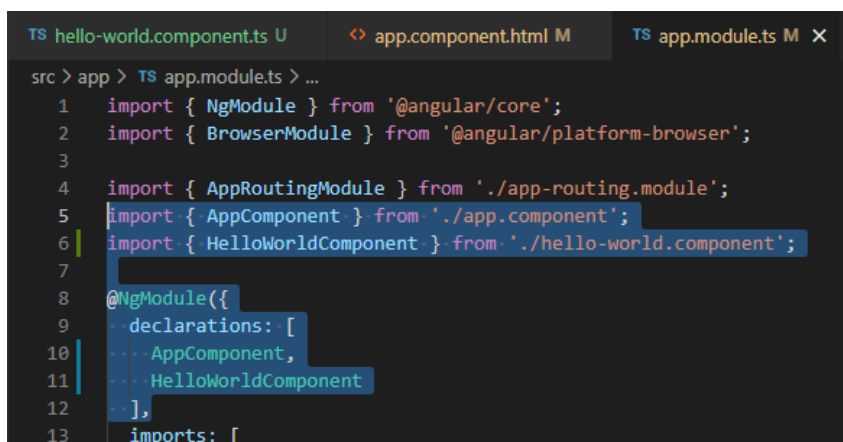
Definiamo un html template, questo può essere inserito direttamente nel componente oppure in un file a parte (`templateUrl`).

```
styles: ['']
```

- 4) Idem per il foglio di stile (`styleUrls`).

```
export class HelloWorldComponent {  
  title = 'Hello World!';  
}
```

- 5) Creazione di una classe dentro cui definire i valori con cui poter rendere dinamici i template



```
TS hello-world.component.ts U  <> app.component.html M  TS app.module.ts M X  
src > app > TS app.module.ts > ...  
1  import { NgModule } from '@angular/core';  
2  import { BrowserModule } from '@angular/platform-browser';  
3  
4  import { AppRoutingModule } from './app-routing.module';  
5  import { AppComponent } from './app.component';  
6  import { HelloWorldComponent } from './hello-world.component';  
7  
8  @NgModule({  
9    declarations: [  
10     - AppComponent,  
11     - HelloWorldComponent  
12    ] ,  
13    imports: [  
14
```

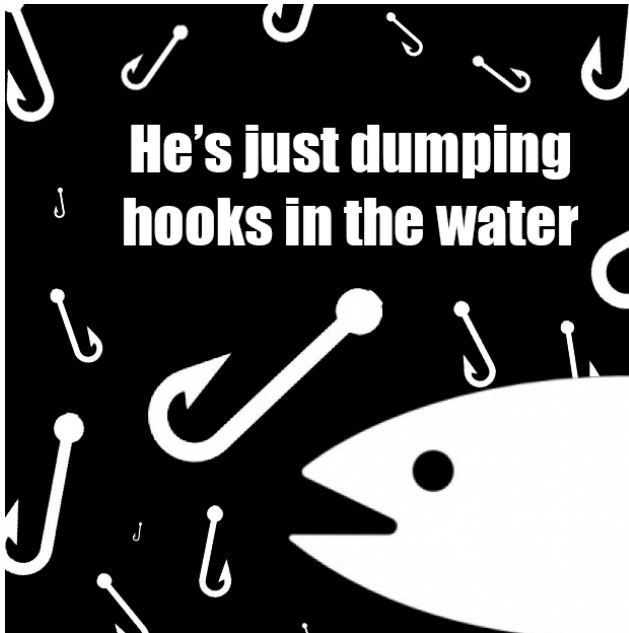
- 6) Import in un NgModule del neo componente

```
<app-hello-world></app-hello-world>
```

- 7) Definizione in `app.component.html`

ng serve per avviare l'app

Capitolo 4 – Lifecycle Hooks



Cosa sono?

Sono specifici **eventi** di un **componente** o di un **directive**.

Es.:

Possono essere usati per inviare dati in un componente dell'applicazione e per molte altre attività.

N.B. : Devono essere **cleanuppati**, affinché non ci sia congestioni legate alla memoria e alle performances dell'app.

ngOnInit → Attivi un Hook

ngOnDestroy → Deattivi un Hook

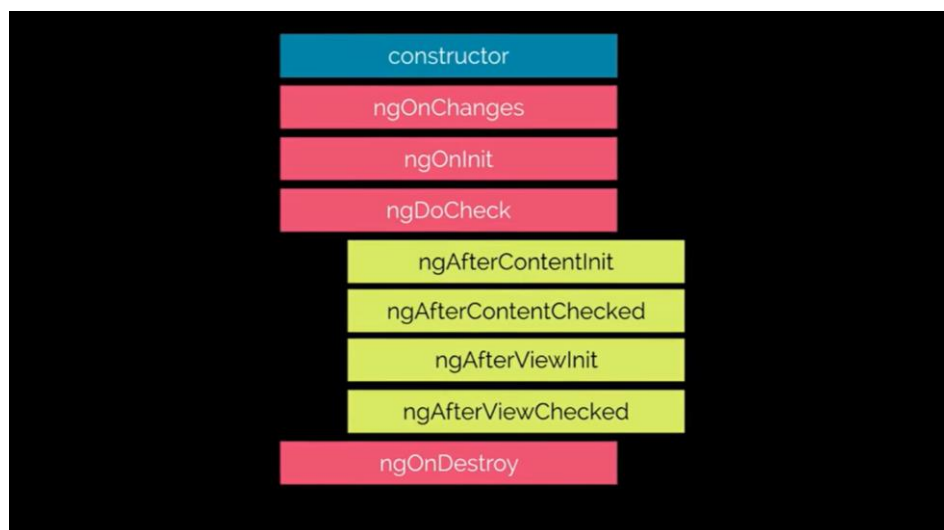


Figura 2 Lista di tutti gli hooks in ordine di inizializzazione

Ma i più realmente usati sono:

- **ngOnInit**
- **ngOnChange**
- **ngOnDestroy**

In pratica...

Importiamo l'hook nel nostro foglio componente

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

...implementiamolo nella nostra classe...

N.B.: **strict:false**, (nel file di configurazione di ts) eviterà un più rigoroso blocco causato da mancate definizioni del tipo.

Capitolo 5 – Text Interpolation

Sistema secondo il quale i dati scritti nel codice scritto in TypeScript viene trasferito al template html

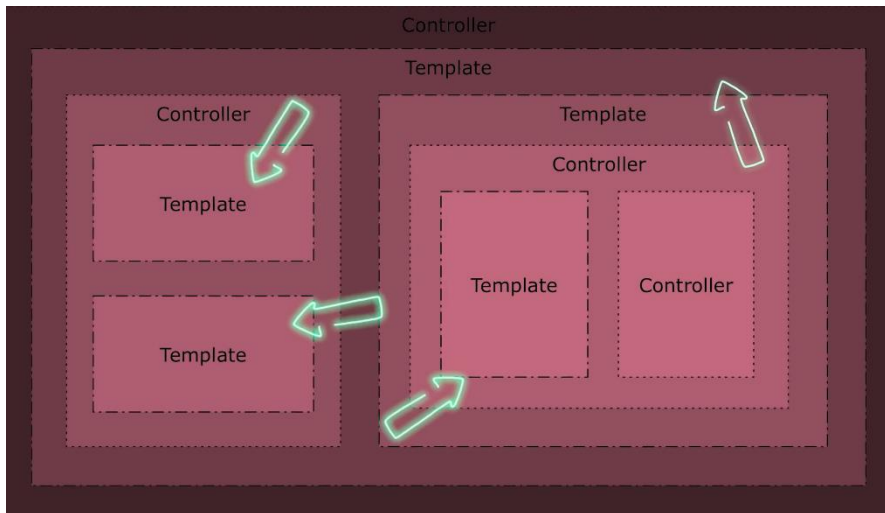
TS Code → HTML Template

Il sistema è il “**Template expression**” ({{}}).

Questo però non supporta:

- =, +=, -=,
- new, typeof, instanceof,
- ++, --
- Espressioni concatenate

Capitolo 6 – Comunicazione tra componenti



Esistono **4 metodi** di **comunicazione** tra i componenti:

- 1) **Binding** (@input e #output).
- 2) **Reference** (@ViewChild e #ContentChild).
- 3) **Provider** (service).
- 4) **Template Outlet**.

Metodo 1 – Input Decorator

Parent → Child

Generiamo un componente padre ed uno figlio

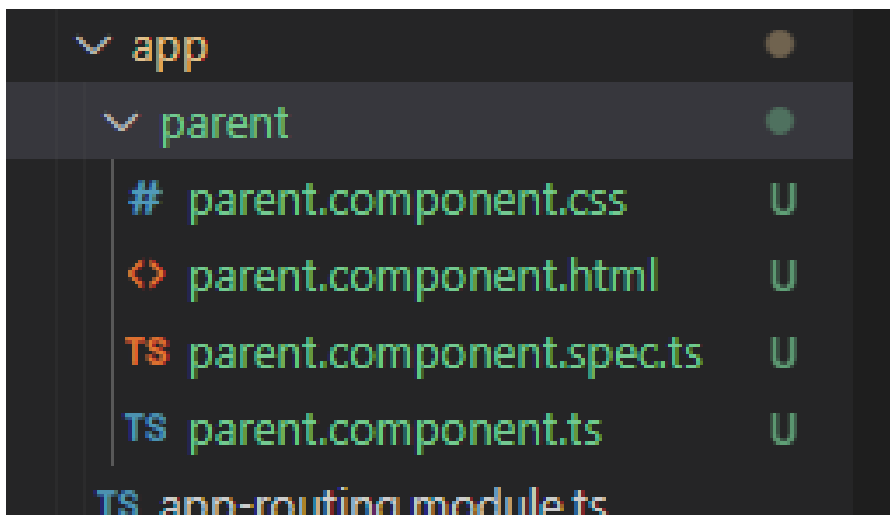


Figura 3 Questo è un componente, generato usando il comando `ng g c nome_componente`

```

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

child.component.html U X
src > app > child > child.component.html > h1
1 <h1>Say {{childMessage}}</h1>

parent.component.html U X
src > app > parent > parent.component.html > app-child
1 <app-child [childMessage] = "Hello from parent"></app-child>

```

Figura 4 Definiamo Input nel figlio e creiamo un template che poi, al momento dell'inserimento del componente figlio in quello padre, verrà riempito con un'informazione

Metodo 1 – Output Decorator

Child → Parent

L'output deve avere un **EventEmitter**, una classe nel core di angular che emette eventi

```

TS child2.component.ts U X
src > app > child2 > TS child2.component.ts > Child2Component
1 import { Component, OnInit, Output, EventEmitter } from '@angular/core';
2
3 @Component({
4   selector: 'app-child2',
5   templateUrl: './child2.component.html',
6   styleUrls: ['./child2.component.css']
7 })
8 export class Child2Component implements OnInit {
9
10  @Output() messageEvent = new EventEmitter<string>();
11
12  constructor() {}
13
14  ngOnInit(): void {
15  }
16
17  // Emette il messaggio
18  sendMessage() {
19    this.messageEvent.emit('Hello from Child2');
20  }
21
22 }
23

child2.component.html U X
src > app > child2 > child2.component.html > button
1 <button (click) = "sendMessage()">5</button>

parent.component.ts U X
src > app > parent > TS parent.component.ts > ParentComponent
1 import { Component, OnInit } from '@angular/core';
2
3 @Component({
4   selector: 'app-parent',
5   templateUrl: './parent.component.html',
6   styleUrls: ['./parent.component.css']
7 })
8 export class ParentComponent implements OnInit {
9
10  constructor() {}
11
12  ngOnInit(): void {
13  }
14
15  // Riceve il messaggio
16  receiveMessage(msg) {
17    alert(msg);
18  }
19
20 }
21

parent.component.html U X
src > app > parent > parent.component.html > app-child2
1 <app-child2 (messageEvent) = "receiveMessage"
2
3
4 <app-child [childMessage] = "Hello from par

```

Figura 5 Uso di event emitter per il passaggio di un valore

Metodo 2 – viewChild

Child → Parent

```

TS child.component.ts U X
src > app > child > TS child.component.ts > ChildComponent
1 import { Component, OnInit, Input } from '@angular/core';
2
3 @Component({
4   selector: 'app-child',
5   templateUrl: './child.component.html',
6   styleUrls: ['./child.component.css']
7 })
8 export class ChildComponent implements OnInit {
9
10  message = 'message from child!';
11
12
13
14
15
16

TS parent.component.ts U X
src > app > parent > TS parent.component.ts > ParentComponent
1 import { Component, OnInit, ViewChild, AfterViewInit } from '@angular/core';
2 import { ChildComponent } from '../child/child.component';
3
4 @Component({
5   selector: 'app-parent',
6   templateUrl: './parent.component.html',
7   styleUrls: ['./parent.component.css']
8 })
9 export class ParentComponent implements OnInit {
10
11  @ViewChild(ChildComponent) child;
12
13  ngAfterViewInit() {
14    alert(this.child.message)
15  }
16

```

Figura 6 Esportazione di un valore che dal Child al Parent

N.B.: Sembra carino!

Capitolo 7 – Stile Componenti

Esistono **3 metodi** per importare lo stile:

- 1) Settando `style` o `styleUrls`
- 2) **Inline**, nel template html
- 3) Con gli **import** CSS

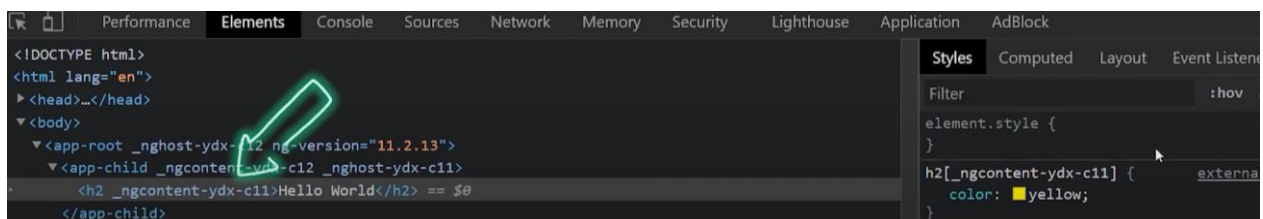
Esempi:

```
styleUrls: ['./child.component.css'],
style: [`h1 {color: blue}`]
```

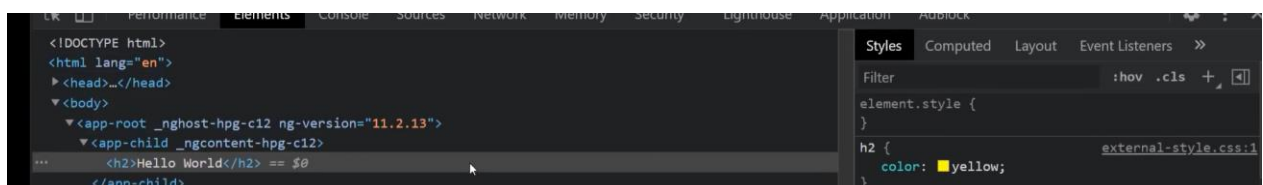
```
TS child.component.ts U # child.component.css U ●
src > app > child > # child.component.css
1 @import '../assets/external-style.css';
```

```
<> child3.component.html U X
src > app > child3 > <> child3.component.html >
1 <style>
2   p {
3     color: orange;
4   }
5 </style>
6 <p>child3 works!</p>
```

Pro-tip: nel render del DOM vi è un incapsulamento dello stile in delle classi specifiche, questo è possibile disabilitarlo



```
styleUrls: ['./child.component.css'],
encapsulation: ViewEncapsulation.None
```



*Esistono dei selettori speciali in css:

Es.: :host o :host-context

Capitolo 8 – NG Content

Cosa è?

Sistema che permette la “**proiezione**” di un componente html in un altro componente.

Creiamo un ng-content nel nostro componente figlio

```
<ng-content [question]></ng-content>
<ng-content [answer]></ng-content>
```

Proiettiamolo dentro il nostro componente root

```
<app-child>
  Hello from the root:
  <h3 question>Gli Ng-Content sono belli?</h3>
  <h4 answer>Diavolo, no!</h4>
</app-child>
```

Usando dei tag per differenziarli.

Capitolo 9 – Template Statements

Cosa è?

È un sistema di risposta al trigger di un evento

N.B.: il contesto del nostro template statement è **localizzato** alla classe del nostro componente o al nostro html template, non è globale.

```

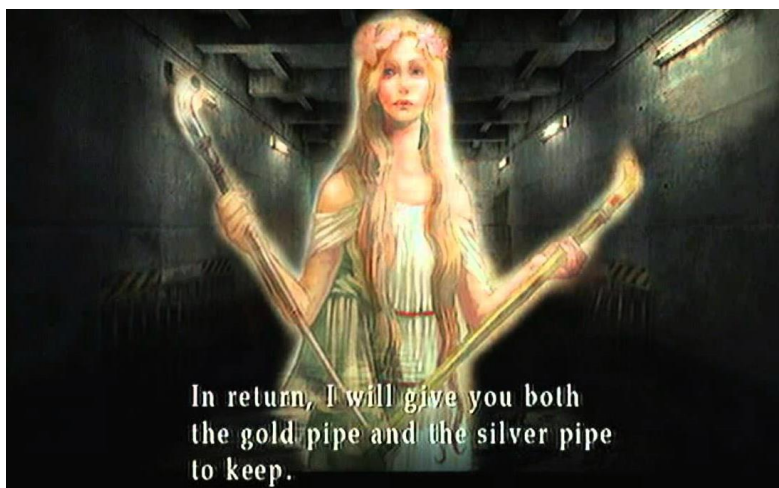
TS app.component.ts M X
src > app > TS app.component.ts > AppComponent > toggleText
9 export class AppComponent /* implement
10
11 title: string = 'Testo titolo';
12
13 showText = false;
14
15 toggleText(event) : void {
16   this.showText = !this.showText;
17   console.log(event)
18 }
19

app.component.html M X
src > app > app.component.html > ...
1 <h1>{{title}}</h1>
2 <app-parent></app-parent>
3
4 <app-child>
5   Hello from the root:
6   <h3 question>Gli Ng-Content sono belli?</h3>
7   <h4 answer>Diavolo, no!</h4>
8 </app-child>
9
10 <button (click) = 'toggleText($event)'>Change Text</button>
11 {{showText}}

```

Figura 7 Quindi metodi che implicano oggetti globali, come document o window non saranno utilizzabili (es. console.log)

Capitolo 10 – Pipe



Cosa sono?

Semplici **funzioni** che possono essere usati nel **template** per accettare un valore in **input** e restituirne uno formattato in **output**.

Ne esistono **vari tipi di integrati** in Angular:

- DatePipe
- UpperCasePipe
- CurrencyPipe
- PercentPipe, ecc...

Implementazione:

```

export class AppComponent /* implements OnI

title: string = 'Testo titolo';

todayDate = new Date();

```

Figura 8 Impostiamo una data

```
<> app.component.html M X
src > app > <> app.component.html > h4
18
19 | <h3>Data di oggi: </h3><h4>{{todayDate | date : 'short'}} </h4>
```

Figura 9 Usiamo una pipe per formattare la data

Pro-tip: Esistono pipes per ogni gusto, scaricale tutte!

Capitolo 11 – Property Binding

Class TS -----> HTML Template

Passa un dato, o setta una proprietà, dalla classe al template html

```
<> app.component.html M X
src > app > <> app.component.html > ...
20
21 | <img [src]="itemImageUrl">

TS app.component.ts M X
src > app > TS app.component.ts > AppComponent >
21
22 | itemImageUrl = '../assets/scroll.jpg'
```

ATTENZIONE: La **text interpolation** e la **property binding** sono confondibili, ma in realtà sono distinte.

Capitolo 12 – Attribute, Class e Style bindings

- 1) Attribute binding
- 2) Calls binding
- 3) Style binding

Attribute Binding:

Permettono di settare valori o attributi direttamente

Per definirla bisogna dichiararla con il prefisso **attr**.

```
<p [attr.nome-attributo]='Sono un espressione'> Paragrafo con attributo binded dentro</p>
```

Class Binding:

Permettono di settare valori o classi direttamente

Per definirla bisogna dichiararla con il prefisso **class**.

```
<p [class.nome-classe]='In Saldo'> Paragrafo con classe binded dentro</p>
```

Se non su true non verrà inserita.

Può essere inserita con stringhe, oggetti o array.

Style Binding:

Permettono di settare valori o stili direttamente

Per definirla bisogna dichiararla con il prefisso **style**.

```
<p [style.background-color]='green'> Paragrafo con stile binded dentro</p>
```

Molti stili:

```
<p [style]='background-color: red; color: white'> Paragrafo con molti stili binded dentro</p>
```

Capitolo 13 – Event Binding

Gestire gli eventi su Angular

Sintassi:

(evento)="funzione"

Esempio:

```
<button (click)="onSave()"></button>
```

Comuni gestori di evento:

Le operazioni classiche che l'utente può compiere sui diversi elementi che costituiscono l'interfaccia dell'applicazione, sono oltre al click o tocco, il posizionamento del mouse sopra all'elemento, la pressione del tasto enter, la pressione di un tasto etc. Ognuno di questi eventi può essere intercettato con i seguenti gestori:

(click) -> intercetta il click su di un elemento

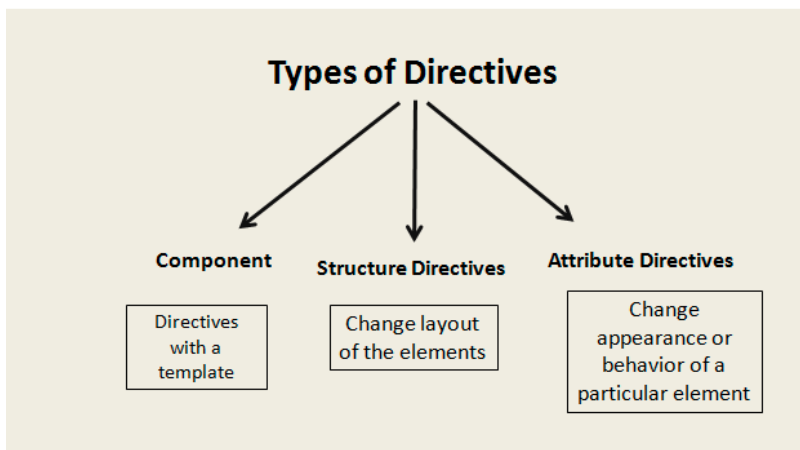
(mouseover) -> intercetta il posizionamento del mouse sopra all'elemento

(input) -> quando l'utente inserisce un valore all'interno di un campo di input

(keyup) -> intercetta la pressione e rilascio di un tasto della tastiera

(keyenter) -> intercetta la pressione del tasto enter della tastiera

Capitolo 14 – Directives



Cosa sono?

Funzioni

Ne esistono **3 tipologie**:

- 1) **Component**,
Con un template
- 2) **Attribute**,
Cambiano l'aspetto di un elemento
- 3) **Structural**,
aggiungono/tolgono elementi al DOM

1) Component Directive

- [NgClass](#)—aggiunge ed elimina un set di CSS classes.

src/app/app.component.html

```
<!-- toggle the "special" class on/off with a property -->
<div [ngClass]="isSpecial ? 'special' : ''">This div is special</div>
```

Es.:

```
<h2 [ngClass]="{'big' : size == 'big', 'small' : size == 'small'}">Testo di prova per sizing</h2>
// Uso di un class binding per modificare una classe in base alla variazione di un valore. In questo classe
varierà in big se il valore size sarà uguale a big, mentre muterà in small se size sarà uguale a small.
```

- [NgStyle](#)—aggiunge e elimina un set di HTML styles

src/app/app.component.html

```
<div [ngStyle]="currentStyles">
  This div is initially italic, normal weight, and extra large (24px).
</div>
```

Es.:

```
<h2 [ngStyle]="{'color': variableColor == 2 ? 'blue' : 'green'}">Testo per ngStyle</h2>
// se variableColor è uguale a 1 il colore dell'elemtno sarà blue, sennò sarà verde. In questo caso, siccome

variableColor: number = 1;
// ...la varibaile è diversa da 1, il colore sarà verde
```

- [NgModel](#)—aggiunge two-way data binding ad un form HTML.

src/app/app.component.html (NgModel example)

```
<label for="example-ngModel">[(ngModel)]:</label>
<input [(ngModel)]="currentItem.name" id="example-ngModel">
```

Allo stesso modo dell'uso del #valore per inviare un valore nella classe, dalla view, è possibile usare questa directive per fare la stessa cosa, ma in modo più pulito, usando la two-way binding, ma senza incasinare tutto.

Es.:

```
<input type="email" [(ngModel)]="email" (keyup)="sendEmail()" placeholder="inserisci l'email">
```

-

```
email = 'test@gmail.com';
sendEmail() {
  console.log('Email ' + this.email + ' re-inviata e modificata');
}
```

2) Structural Directive

▪ ngFor

Ciclo for (of) per Angular.

Es.:

```
<li *ngFor="let hero of heroes; let i = index"><h3>{{i + 1}} {{hero.name}}</h3></li>
// Stamp dei nomi dell'array posto nel componente
```

▪ ngIf

Ciclo if per Angular

Es.:

```
<ul *ngFor="let lucisingole of lucidb">
  <li *ngIf="lucisingole.op">Luce accesa</li>
</ul>
```

// Render dei soli li, della lista "lucidb" che hanno un valore true

▪ ngSwitchCase

Switch-Case per Angular

Es.:

```
luminosita: any[] = [
  {ill:0},
  {ill:1},
  {ill:2}
];
```

// Array di oggetti in classe componente

```
<div>
  <ul *ngFor="let lum of luminosita" [ngSwitch]="lum.ill">
    <li *ngSwitchCase="0" style="opacity: 0.1"><h4>Buia</h4></li>
    <li *ngSwitchCase="1" style="opacity: 0.5"><h4>Penombra</h4></li>
    <li *ngSwitchCase="2" style="opacity: 1"><h4>Illuminata</h4></li>
  </ul>
</div>
```

// Uso Switch-Case in html template

3) Attribute Directive

▪ Non approfondite



Figura 10 Ma magari non troppo...

Uso di un web module

Nel qual caso in cui ci fosse il bisogno di inserire un **dato** in un form, ed inviarlo alla classe del nostro componente, è possibile usare la notazione **#dato**, che definisce la **variabile locale del template**.

Essa invierà il dato alla classe, permettendo una sua eventuale elaborazione.

Es.:

```
<h3>Inserisci valore</h3>
<input #dato placeholder="Inserisci un dato">
<button (click)="invioDati(dato.value)">Invia</button>
// Template HTML
```

```
invioDati(dato) {
  console.log(dato + ' ' +'arrivato')
}
// Classe componente
```

Capitolo 15 – Form

- Form **non reattivo**:
Un normalissimo form

Es.:

```
<select (change)="selectedOption(selected)" #selected>
  <option value="1">Saluta con il log</option>
  <option value="2">Saluta con l>alert</option>
</select>
```

- Form **reattivo**:
Uso di FormControl

`[(ngModel)]` = "someValue"



= "someValue"

Nota: Per il passaggio di valore alla classe è possibile l'uso di **ngModel**, usando la sintassi "**banana in the box**"

Capitolo 16 – Richieste HTTP

Chiamate http ad una API

- 1) Import del modulo http nel modulo

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

- 2) Istanziamento del servizio http nel costruttore

```
constructor(public http:HttpClient) {}
```

- 3) Creazione di una funzione **Observable**, atta ad effettuare la chiamata ed attendere una risposta che...

```
loadUsers(): Observable<Object> {
  let url = `https://pokeapi.co/api/v2/pokemon/1`;
  return this.http.get(url)
}
```

4) Verrà processata da una **callback** che processerà il dato

```
pokemonName: string;
callback = (response) => {
  console.log(response)
  this.pokemonName = response.name
}
```

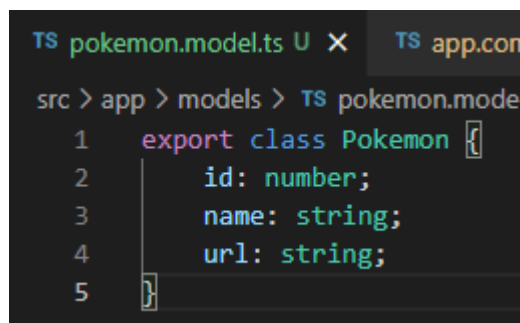
5) Avviare all'inizializzazione del componente? **ngOnInit** è la risposta.

```
ngOnInit() : void{
  this.loadUsers().subscribe(this.callback);
}
```

Capitolo 17 – Model

Model: Astrazione di una tabella del Database atta a strutturare il dato secondo un modello predefinito dallo sviluppatore.

Es.



```
TS pokemon.model.ts U X TS app.com
src > app > models > TS pokemon.model
1  export class Pokemon {
2    id: number;
3    name: string;
4    url: string;
5  }
```

Figura 11 Model di Pokemon in cartella Models

Utilizzo:

Istanziamento in array della lista che otterremo in get, dopo aver effettuato una chiamata

```
pokemons: Pokemon[] = new Array();
// La forma che otterrà ogni item della lista ottenuta assumerà la forma del model creato

return this.http.get<Pokemon[]>(urlPokemon)
// Ciò che verrà restituito dall'Observable verrà trasfigurato grazie al model importato

this.pokemons = response.results
// Nella nostra callback definiremo l'array in base alla response in arrivo.
```

Capitolo 18 – Routing



Routing: Capacità di impostare più pagine, renderizzando, condizionalmente, i componenti in base al path url voluto.

Figura 12 Choice your route!

Definizione del **route**, esplicitando il componente da renderizzare in base al path url richiesto

```
TS app-routing.module.ts M X TS app.component.ts M <> app.component.html M
src > app > TS app-routing.module.ts > ...
1 import { NgModule } from '@angular/core';
2 import { RouterModule, Routes } from '@angular/router';
3 import { RoutingtestComponent } from '../routingtest/routingtest.component';
4
5 const routes: Routes = [{
6   path: 'tutorial',
7   component: RoutingtestComponent
8 }];
9
```

Figura 13 Es. in `http://localhost:4200/tutorial`

Componente da inserire nel **componente root** che abilita l'uso del **routing**.

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

Gestione **href** in base al routing (menu rudimentale)

```
<li><a href="javascript:void(0)" [routerLink]="['/']">Homepage a </a></li>
<li><a href="javascript:void(0)" [routerLink]="['/tutorial']">Pagina /tutorial</a></li>
```

Per quanto riguarda approfondimenti su uso di **path dinamici** (:id), **reindirizzamenti** (pathMatch) e gestione di **path errati** (path: **), rivedere corso.

Capitolo 19 – Service

Metodologia, sfruttante l'OOP, atta al riutilizzo di una stessa classe in più componenti distinti.

Il sistema permette il riutilizzo di funzioni, evitando la riscrittura di codice uguale in componenti distinti.

- 1) Creiamo un file distinto con estensione **service.ts**.
- 2) Dichiariamo che la classe sarà injectable, ovvero iniettabile in altri componenti

```
@Injectable({  
  providedIn: 'root'  
})
```

// Specificiamo **root** per avere una **iniettabilità** in ogni componente

- 3) Creiamo la classe con il metodo da riutilizzare

```
export class TutorialService {  
  salutFromService() {  
    console.log('Ciao, sono una funziona sfrittata in più componenti!')  
  }  
}
```

- 4) Importiamolo e istanziamolo nel costruttore del componente in cui vogliamo usare il servizio

```
constructor(public tservice: TutorialService ) {
```

- 5) Inizializzazione del servizio al primo rendering della pagina

```
ngOnInit(): void {  
  this.tservice.salutFromService();  
}
```

Fonti:

- <https://www.youtube.com/watch?v=3dHNOWTI7H8>
- <https://www.youtube.com/watch?v=AAu8bjj6-UI> (sconsigliato)
- <https://angular.io/docs>

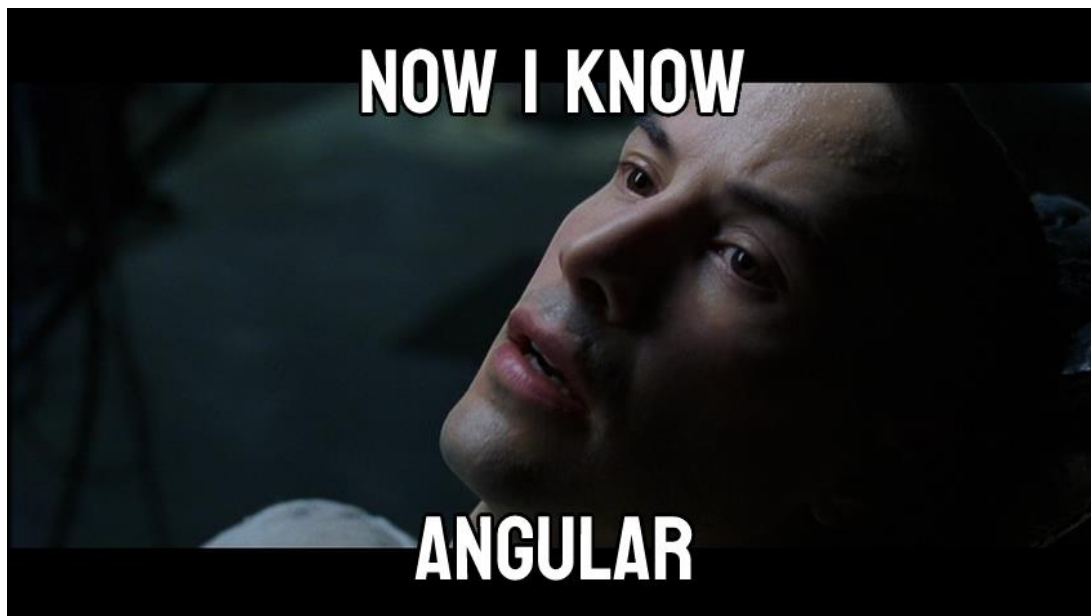


Figura 14 ...circa.