Documentazione Angular

V1.0.0

Federico Perin - f.perin@reply.it

Maggio 2025

Table of contents

1. Angular 4
1.1 Angular MVC 4
1.2 Decoratori 6
2. Setup 7
3. NgModule 8
3.1 Struttura 8
3.2 forRoot( ) 9
3.3 forChild( ) 10
4. Component 12
4.1 Templates e views 14
4.2 Sintassi Template 15
4.2.1 Direttive Build-In 16
4.2.2 Data Binding 19
4.3 Ciclo di vita di un componente (Lifecycle) 22
4.4 ng-template 23
4.5 ng-content 23
4.6 ng-container 23
5. Pipes 25
5.1 Utilizzo 25
5.2 Pipes Custom 26
5.3 Pipes Pure 26
6. Services 27
6.1 Dependency Injection 27
6.2 Injectables 28
6.3 Providers 28
6.3.1 useClass 29
6.3.2 useExisting 29
6.3.3 useFactory 30
6.3.4 useValue 31
6.4 Injector 33
7. Routing 34
7.1 Impostare il routing 34
7.2 RouterLink 36
7.3 Path params 36
7.4 Nesting routes 37

7.5 Guardie 38
7.6 Lazy loading 39
8. Observables 40
8.1 RxJS 41
8.1.1 Subject, BehaviorSubject e ReplaySubject 42
8.2 Pipe( ) 43
9. Direttive personalizzate 44
10. Template-driven form 45
10.1 Costruzione 45

1. Angular

Angular è un framework Front-End per lo sviluppo di Single Page Application (SPA), creato e mantenuto da Google. Nato come evoluzione di AngularJS, Angular è progettato per facilitare la creazione di applicazioni moderne, scalabili e manutenibili. Grazie a una struttura basata su componenti, una potente gestione delle dipendenze e strumenti integrati per il routing, il form handling e la comunicazione con API, Angular permette agli sviluppatori di costruire interfacce utente dinamiche e performanti.

Angular si ispira al pattern architetturale MVC (Model-View-Controller), riorganizzandolo in una forma più moderna e component-based. In Angular, la View è rappresentata dal template HTML, il Model è costituito dai dati e dagli oggetti gestiti dai componenti e dai servizi, mentre la Controller logic è incorporata direttamente nei componenti attraverso il codice TypeScript. Questa struttura favorisce una netta separazione delle responsabilità e rende il flusso dei dati chiaro e controllato.

Ogni componente Angular è associato a un file HTML, che definisce la struttura visiva (il template) dell'interfaccia, e a file CSS o SCSS, che gestiscono gli stili e l'aspetto grafico. Questa separazione chiara tra logica, struttura e presentazione aiuta a mantenere il codice organizzato e facilmente manutenibile.

Per la logica di business viene utilizzato TypeScript, un superset di JavaScript che introduce la tipizzazione statica, il supporto a classi e interfacce, e avanzate funzionalità di tooling. L'utilizzo di TypeScript migliora la qualità del codice, facilita la manutenzione e consente di sfruttare strumenti di sviluppo più potenti.

Note

In questo documento verranno trattate le versioni 7 e 12 di Angular, in quanto compatibili con il framework utilizzato dal cliente XDCE.

Info

Documentazione Angular 7
Documentazione Angular 12

1.1 Angular MVC

Il Model-View-Controller (MVC) è un design pattern tipicamente utilizzato per lo sviluppo di web applications perché permette la "separation of concerns" in cui il data model viene separato dalla business e presentation logic.

Nel seguente schema viene riportato come MVC è stato riorganizzato in una forma component-based.

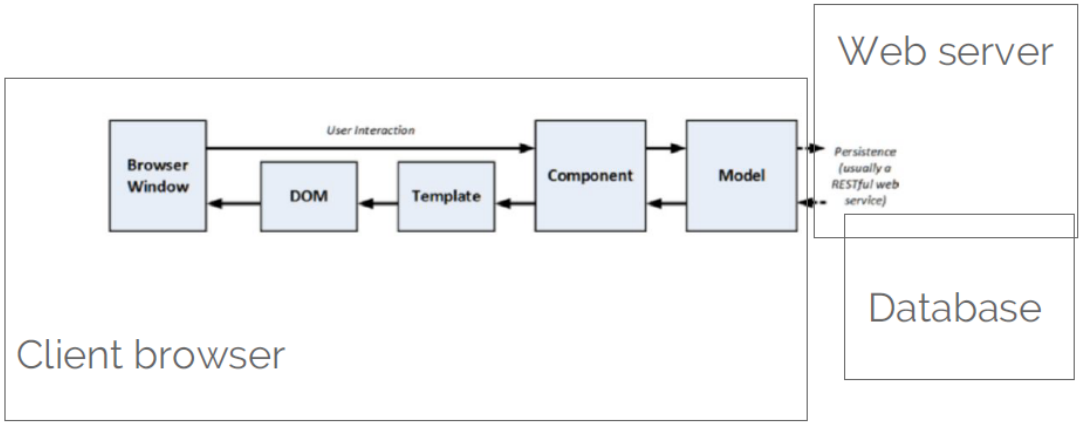

Figura 1: Schema MVC in chiave Angular

All'interno del cliente troviamo gli elementi: Model, Component, Template, DOM e Browser Window.

Il Model è il livello che sa cosa rappresentano i dati e come gestirli, ma non si occupa di come vengono mostrati o come vengono interagiti dall'utente. Perciò, risiedono i dati dell'applicazione e i servizi che offrono la logica per recuperare e manipolare i dati oltre a fornire chiamate HTTP a API. Inoltre, possono risiedere enum o interfacce che rappresentano la struttura dei dati.

> **Note**
>
> Non deve esporre i dettagli su come i dati sono ottenuti o gestiti (il web service non deve essere esposto al controller e alle view).
> Non deve contenere la logica che trasforma i dati rispetto alle interazioni con l'utente (compito del controller).
> Non deve contenere la logica di visualizzazione dei dati (compito della view).

Il Component svolge il ruolo del Controller tradizionale dell'MVC. Ha il compito di gestire la logica dell'interfaccia utente: riceve input dall'utente, reagisce agli eventi (click, input, ecc.), e aggiorna i dati del Model o della View. Percio, il component si occupa di interagire con il Model per fornire dati e comportamenti al template HTML

Il Component in Angular è composto da:

- un file TypeScript (.ts) che contiene la classe con proprietà e metodi → (Controller);
- un file HTML (.html) che definisce la struttura visiva → (View);
- uno o più file CSS/SCSS (.css o .scss) per gli stili → (Styling).

> **Note**
>
> Non deve gestire la visualizzazione dei dati (non deve modificare il DOM).
> Non deve gestire la logica di persistenza dei dati (Compito del modello attraverso il web service).

Il Template è la parte dell'applicazione che definisce come i dati vengono presentati all'utente. Contiene la logica di markup necessaria a presentare i dati all'utente.

Il template è normalmente scritto in HTML, ma arricchito con funzionalità aggiuntive fornite da Angular, come: - Data binding (legare i dati del componente al template) - Strutture di controllo (es. *ngIf,* ngFor) - Event binding (es. (click), (input)) - Pipe per la formattazione dei dati

Queste funzionalità verrano illustarte nei prossimi capitoli.

> **Note**
>
> Non deve contenere alcuna logica di modifica, creazione e gestione dei dati.
> Non deve contenere la logica dell'applicazione.

1.2 Decoratori

Angular sfrutta i decoratori introdotti da TypeScript per "annotare" classi, metodi o proprietà, arricchendoli di metadati e abilitando funzionalità avanzate a runtime.

Un esempio tipico è il decoratore @Component(), che viene applicato a una classe per indicare che rappresenta un componente Angular. Il decoratore fornisce ad Angular tutte le informazioni necessarie, come il selettore, il template e i file di stile associati.

I decoratori possono essere configurati tramite parametri passati al momento della dichiarazione, consentendo una gestione dinamica e personalizzata del comportamento dell'applicazione.

Decorator	Descrizione
@NgModule	Definisce un modulo che contiene componenti, direttive, pipe e providers.
@Component	Dichiara che una classe è un componente e ne fornisce i meta-dati.
@Injectable	Dichiara che una classe ha dipendenze che devono essere iniettate nel costruttore quando il Dependency Injector crea l'istanza della classe.
@Directive	Dichiara che una classe è una direttiva e ne fornisce i metadati.
@Pipe	Dichiara che una classe è una pipe e ne fornisce i metadati.
@Input	Dichiara una proprietà di input che si può aggiornare con un property binding.
@Output	Dichiara una proprietà di output che emette eventi a cui ci si può sottoscrivere con un event binding.
@HostBinding	Effettua il binding di una proprietà dell'elemento host ad un proprietà o direttiva del componente.
@HostListener	Sottoscrizione ad un evento dell'elemento host tramite una direttiva o metodo del componente, eventualmente passando un parametro.
@ContentChild	Esegue una query di selezione dei componenti figli, effettuando il binding del primo risultato ottenuto con una proprietà della classe. Si riferisce ai componenti che sono figli del componente corrente nel DOM in cui esso è utilizzato.
@ContentChildren	Esegue una query di selezione dei componenti figli, effettuando il binding dei risultati ottenuti con una proprietà della classe. Si riferisce ai componenti che sono figli del componente corrente nel DOM in cui esso è utilizzato.
@ViewChild	Esegue una query di selezione dei componenti figli, effettuando il binding del primo risultato ottenuto con una proprietà della classe. Si riferisce ai componenti che sono inseriti nel DOM del template collegato al componente.
@ViewChildren	Esegue una query di selezione dei componenti figli, effettuando il binding dei risultati ottenuti con una proprietà della classe. Si riferisce ai componenti che sono inseriti nel DOM del template collegato al componente.

2. Setup

Per installare e usare Angular 7 o 12 occore prima di tutto installare il gestore di package npm:

- Download Node 10.24.1 per Angular 7
- Download Node 12.22.7 per Angular 12

> **Info**
>
> Per gestire le versioni di node è utile installare NVM.
> Scarica nvm-setup.exe per installare npm.
> Verifica con:
>
> - `nvm version` per verificare se NVM è stato installato correttamente.
> - `npm -v` per verificare se npm è stato installato correttamente.

Si passa poi a installare la Angular CLI:

```
npm install -g @angular/cli
```

o

```
npm install -g @angular/cli@12
```

e verificare con il commando `ng version` per verificare la corretta installazione.

Per creare un progetto Angular si utilizza:

```
ng new nome-progetto
```

e per avviare il progetto:

- spostiamoci nella directory con `cd nome-progetto`,
- `ng serve` per lanciare il server Angular,
- collegarsi al http://localhost:4200 nel browser.

3. NgModule

Le applicazioni Angular sono composte da blocchi chiamati moduli o NgModules. Un modulo in Angular è una raccolta logica di componenti, direttive, pipe e servizi, raggruppati nello stesso contesto di compilazione. Un NgModule può inoltre importare funzionalità da altri moduli ed esportare le proprie, rendendole disponibili ad altre parti dell'applicazione. Ne conseguono i vantaggi di una chiara organizzazione dell'applicazione e dalla posibilità di riuso del codice.

> **Note**
>
> Le principali librerie integrate di Angular sono organizzate in NgModules, ad esempio: - `HttpClientModule` da `@angular/common/http` - `FormsModule` da `@angular/forms`

3.1 Struttura

Ogni applicazione Angular ha almeno un modulo principale, detto root module, che per convenzione viene chiamato AppModule. Il root module definisce come effettuare il bootstrapping dell'applicazione, ovvero quali componenti "root" inserire inizialmente nella pagina.

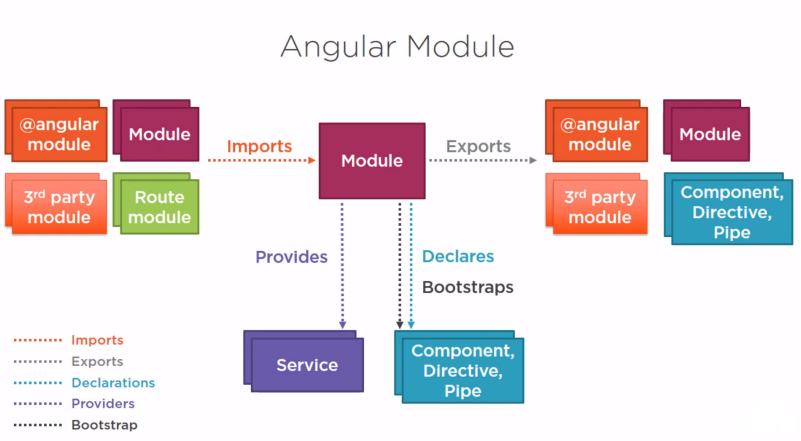

Figura 2: Schema Angular Module

Un modulo Angular viene definito tramite una classe decorata con il decorator `@NgModule`. Questo decorator riceve come parametro un oggetto che descrive i metadati del modulo:

- `declarations`: i componenti, i servizi, le direttive e le pipe dichiarate all'interno del modulo;
- `exports`: componenti, direttive o pipe che il modulo rende disponibili ad altri moduli;
- `imports`: gli altri moduli i cui componenti o servizi sono necessari a questo modulo;
- `providers`: i servizi disponibili tramite dependency injection all'interno dell'applicazione.
- `bootstrap`: usato solo nel root module, specifica il componente principale che Angular deve creare e inserire nel DOM quando viene avviata l'applicazione.

Note

Affinché il browser possa accedere al codice contenuto nei moduli importati, è necessario specificare dove si trovano questi moduli. A questo scopo si utilizzano i moduli TypeScript, che descrivono l'associazione tra gli oggetti esportati e i file nei quali il codice è stato definito.
Per esempio:

```
import { CommonModule } from '@angular/common';
```

Example

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';
import { AppComponent } from './app.component';
import { LoggerService } from './logger.service';

@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    CommonModule
  ],
  providers: [
    LoggerService
  ],
  exports: [
    AppComponent
  ]
})
export class AppModule { } //Angular module
```

3.2 forRoot( )

Il metodo `forRoot()` permette di dichiarare che un modulo fornisce dei servizi tramite il campo providers, i quali devono essere registrati una sola volta all'interno del modulo principale dell'applicazione. È utile utilizzare `forRoot()` quando si desidera che un servizio di un modulo sia gestito come singleton, ovvero che esista un'unica istanza condivisa in tutta l'applicazione.

Il `forRoot()` è un metodo statico che accetta una configurazione opzionale e restituisce un oggetto di tipo `ModuleWithProviders<T>` che include un array di provider. Tutti i provider registrati attraverso `forRoot()` vengono caricati in modo eager, cioè immediatamente all'avvio dell'applicazione.

Esempio:

Supponiamo di avere un `CustomMenuModule` che fornisce un servizio `MenuService`.

Example

```
//custom-menu.module.ts
@NgModule({
  declarations: [MenuComponent],
  exports: [MenuComponent]
})
export class CustomMenuModule {
  static forRoot(config: MenuConfig): ModuleWithProviders<CustomMenuModule> {
    return {
      ngModule: CustomMenuModule,
      providers: [
        { provide: MenuConfig, useValue: config },
        MenuService
      ]
    };
  }
}
```

Il `forRoot()` ritorna l'oggetto `ModuleWithProviders`, che restituisce un modulo insieme a suoi provider, in questo caso `MenuService` e la sua configurazione. Se non viene ritornato un oggetto `ModuleWithProviders`, Angular non sa che insieme al modulo che provider registrare.

Quindi in providers registriamo `MenuService` una sola volta.

Nel `AppModule` importiamo `CustomMenuModule` indicando una configurazione al menù

Example

```
//app.module.ts
imports: [
  CustomMenuModule.forRoot({ theme: 'dark' })
]
```

3.3 forChild( )

Il metodo `forChild()` permette di importare un modulo senza reimpostare anche i provider definiti in `forRoot()`. Permette quindi di utilizzare solo i componenti di un modulo in altre parti dell'applicazione senza duplicare/ridefinire i servizi.

Note

il metodo statico `forChild()` non accetta configurazioni e non registra nuovi provider. Serve solo per importare componenti, direttive, e pipe.

Esempio:

Se vogliamo aggiungere il `forChild()` nel `customerMenu`.

Example

```
//custom-menu.module.ts
static forChild(): ModuleWithProviders<CustomMenuModule> {
  return {
    ngModule: CustomMenuModule,
    providers: [] // Nessun provider
  };
}
```

Dove lo importiamo usiamo il `forChild()`.

Example

```
//app.module.ts
imports: [
  CustomMenuModule.forChild()
]
```

Il `forChild()` risulta utile per due motivi principali:

- Definire le rotte nei moduli secondari: Per esempio, si immagini di avere un'applicazione di e-commerce. Nel modulo principale si stabiliscono le rotte principali una sola volta tramite `forRoot()`, come la Home, la pagina dei Prodotti, il Carrello, ecc.. . Nei moduli secondari, invece, si usano le rotte tramite forChild per aggiungere rotte specifiche relative solo a quel modulo. Ad esempio, nel modulo che gestisce i prodotti, si potrebbero definire rotte come il dettaglio del prodotto o la pagina per la modifica di un prodotto.
- Abilitare il Lazy Loading: `forChild()` è anche essenziale per implementare il lazy loading, una tecnica che consente di caricare i moduli solo quando sono effettivamente necessari. Questo permette di ottimizzare le prestazioni dell'applicazione, caricando solo il codice relativo alle funzionalità richieste dall'utente, riducendo così il tempo di caricamento iniziale.

Info

Metodo	Dove si usa	Cosa fa
`forRoot()`	Nel `AppModule` (root module)	Registra servizi/providers (singleton)
`forChild()`	Nei feature module o lazy module	Usa solo componenti/direttive/pipe, senza duplicare servizi

4. Component

Un componente in Angular è un elemento fondamentale che definisce una porzione dell'interfaccia utente, composta da dati e elementi visivi che vengono mostrati nel DOM (vista). Esso incapsula la logica necessaria per gestire l'interazione e il comportamento dell'interfaccia.

Un componente è costituito da una classe che contiene i dati e la logica ad essi associati. Grazie ai metadati definiti dal decoratore `@Component`, il componente è composto da tre elementi principali:

- **Template (HTML)**: Definisce la struttura e la presentazione dell'interfaccia utente. È il codice HTML che viene renderizzato quando il componente viene caricato. Il template include codice HTML e può contenere direttive e binding di dati offerti da Angular, come il binding per proprietà, eventi, o classi CSS dinamiche.
- **Classe (TypeScript)**: Contiene la logica del componente. La classe è definita in TypeScript e gestisce il comportamento, i dati e le interazioni dell'interfaccia utente. Può includere proprietà, metodi e logica per gestire eventi, dati in ingresso e in uscita, e anche per effettuare chiamate API.
- **Stili (CSS/SCSS)**: Ogni componente può avere stili CSS o SCSS associati che ne determinano l'aspetto visivo. Questi stili sono applicati solo al componente specifico, grazie all'incapsulamento degli stili, evitando conflitti con altri componenti dell'applicazione.

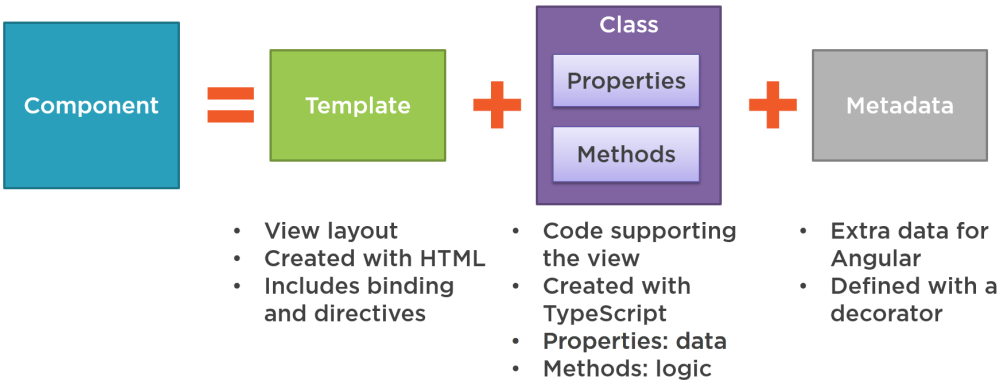

Figura 3: Schema Component

Vediamo un esempio di semplice component:

Example

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

@Component({
  selector: 'app-hello-world',
  templateUrl: './hello-world.component.html',
  styleUrls: ['./hello-world.component.css']
})
export class HelloWorldComponent {

  public message: string;

  constructor(){
    this.message = 'Ciao mondo! Benvenuto su Angular!';
  }

}
```

Grazie al decoratore `@Component` possiamo definire il component con i seguenti metadati:

- **selector**: definisce il nome del tag HTML personalizzato che Angular usa per individuare dove istanziare e rendere visibile il componente. Ad esempio, se il template contiene `<app-hello-world></app-hello-world>`, Angular rileva il tag e monta un'istanza di `HelloWorldComponent` in quella posizione.
- **templateUrl**: indica il percorso relativo al file HTML che definisce la vista del componente. In alternativa, è possibile specificare il template direttamente nel codice tramite la proprietà template.
- **styleUrls**: specifica uno o più percorsi ai file CSS (o SCSS) che contengono gli stili applicati solo al componente. Gli stili definiti in questi file si applicano in modo isolato al componente, grazie all'incapsulamento dello stile che Angular applica di default (ViewEncapsulation).

Info

In Angular, **ViewEncapsulation** controlla come gli stili CSS definiti in un componente vengono applicati: - Solo al componente stesso (incapsulamento). - Oppure si propagano anche agli altri elementi HTML (come avviene normalmente nel browser).

L'obiettivo principale è isolare gli stili di un componente, evitando che "inquinino" altri componenti o che vengano sovrascritti da stili esterni.

Angular fornisce tre modalità principali:

Tipo	Descrizione	Effetto pratico
Emulated (predefinito)	Simula l'incapsulamento degli stili. Angular aggiunge attributi HTML personalizzati ai selettori CSS per limitare l'ambito.	Gli stili sono applicati solo al componente e ai suoi figli.
None	Nessun incapsulamento: gli stili sono globali.	Gli stili si propagano ovunque nell'applicazione.
ShadowDom	Usa direttamente il vero Shadow DOM del browser (se disponibile).	Gli stili sono naturalmente isolati grazie al browser.

Il ViewEncapsulation puo essere configurato nella proprieta encapsulation di `@Component`:

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

@Component({
  selector: 'app-example',
  templateUrl: './example.component.html',
  styleUrls: ['./example.component.css'],
  encapsulation: ViewEncapsulation.Emulated // (default)
})
export class ExampleComponent { }
```

Non è stato riportato nell'esempio del component `HelloWorldComponent`, esiste anche la proprieta `providers` che permette di dichiarare i servizi di cui il componente ha bisogno. Angular crea un'istanza locale di quel servizio solo per quel componente e per i suoi figli. Questo meccanismo è alla base del sistema di Dependency Injection di Angular. Approfondiremo l'argomento nel capitolo dedicato ai servizi.

Note

In Angular esiste la convenzione di dare come nome della classe del componente usando il PascalCase: I nomi delle classi devono iniziare con una lettera maiuscola e ogni nuova parola deve iniziare con una maiuscola.

Esempio: `HelloWorldComponent`

per coerenza il nome della classe deve riflettere il nome del file in kebab-case: I selettori devono essere in minuscolo e separati da trattini.

File: `hello-word.component.ts`

Analogamente anche il selettore deve essere in kebab-case aggiungendo un prefisso.

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

solitamente si utilizza il prefisso app- ma a seconda delle norme interne dell'azienda puo essere stabilito un altro dipo di prefisso.

4.1 Templates e views

In Angular, le viste non sono necessariamente costituite da un singolo componente con il suo template, ma è altamente consigliato adottare una struttura gerarchica di componenti, che possono formare l'intera vista o semplicemente un singolo widget. Questa gerarchia permette di gestire in modo efficiente la visibilità e la modifica di sezioni o pagine dell'interfaccia utente, trattandole come un'unica entità coesa.

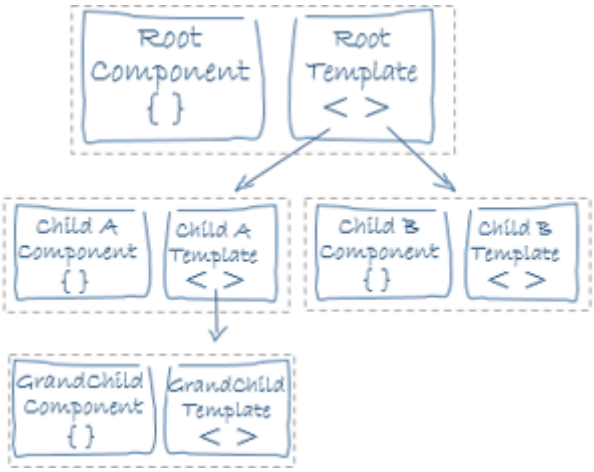

Figura 4: Schema gerarchia di components

Come si può osservare nell'immagine precedente, il template radice è costituito da un insieme di componenti figli, che a loro volta possono includere altre viste, dando così origine a una gerarchia di componenti.

> **Note**
>
> Una gerarchia di viste può comprendere componenti appartenenti allo stesso NgModule, ma anche componenti definiti in NgModule differenti.

4.2 Sintassi Template

Un template è simile a un normale HTML, ma include anche la sintassi specifica di Angular, che modifica l'HTML in base alla logica dell'applicazione e allo stato dei dati del DOM. Nel template puoi utilizzare il data binding per sincronizzare i dati dell'applicazione con quelli del DOM, le pipe per trasformare i dati prima che vengano visualizzati, e le directive per applicare la logica dell'applicazione a ciò che viene mostrato.

Vediamo un esempio di template:

Example

```
  <!-- hero-list.component.html -->
  <div class="hero-list">
    <h2>Lista degli Eroi</h2>

    <!-- Interpolazione (data binding) -->
    <p>Benvenuto, {{ user.name }}!</p>

    <!-- Data binding per l'attributo -->
    <img [src]="user.imageUrl" alt="{{ user.name }}">

    <!-- Diretiva condizionale -->
    <div *ngIf="heroSelected">
      <p>L'eroe è selezionato!</p>
    </div>

    <ul>
      <!-- Event binding -->
      <li *ngFor="let hero of heroes" (click)="selectHero(hero)">{{ hero.name }}</li>
    </ul>

    <!-- ngSwitch per visualizzare informazioni in base all'eroe selezionato -->
    <div [ngSwitch]="heroSelected.name">
      <div *ngSwitchCase="'Iron Man'">
        <h3>Iron Man</h3>
        <p>Iron Man è un genio miliardario, playboy e filantropo che combatte per la giustizia!<
/p>
      </div>
      <div *ngSwitchCase="'Spider-Man'">
        <h3>Spider-Man</h3>
        <p>Spider-Man è un giovane eroe che utilizza i suoi poteri per proteggere New York!</p>
      </div>
      <div *ngSwitchDefault>
        <h3>Eroe Non Selezionato</h3>
        <p>Seleziona un eroe dalla lista per visualizzare ulteriori dettagli!</p>
      </div>
    </div>

    <!-- Componente Figlio con Two-Way Data Binding -->
    <app-hero-detail [hero]="heroSelected" (heroSelected)="onHeroSelectedDetails($event)"></app-
hero-detail>
  </div>
```

Nelle seguenti sezioni verrano illustarte tutti i nuovi costrutti di Angular.

4.2.1 Direttive Build-In

In Angular, esistono delle direttive built-in, ossia direttive predefinite fornite direttamente dal framework, che permettono di controllare il comportamento, l'aspetto o la struttura del DOM in modo dinamico. Sono strumenti molto potenti perché consentono di modificare la pagina HTML in base allo stato dell'applicazione senza dover manipolare direttamente il DOM con JavaScript.

Possiamo dividere le direttive built-in in due grandi categorie: - direttive strutturali: - *ngIf* - ngSwitch - *ngFor - direttive di attributo: - ngClass - ngStyle

***ngIf**

La direttiva `*ngIf` è una direttiva strutturale built-in fornita da Angular. Permette di aggiungere o rimuovere dinamicamente un elemento dal DOM (Document Object Model) in base a una condizione booleana. Quando la condizione associata a `*ngIf` è vera, l'elemento (o il blocco di contenuto) viene aggiunto al DOM. Al contrario, se la condizione è falsa, l'elemento viene completamente rimosso dal DOM.

Quando un elemento viene rimosso tramite `*ngIf`, non viene semplicemente nascosto, ma viene effettivamente eliminato dal DOM. Di conseguenza, ogni volta che la condizione cambia e l'elemento viene reinserito, viene creato nuovamente, perdendo qualsiasi stato precedente.

Example

```
<div *ngIf="heroSelected">
  <p>L'eroe è selezionato!</p>
</div>
```

ngSwitch

La direttiva `ngSwitch` è una direttiva strutturale di Angular che permette di gestire la visualizzazione condizionale di blocchi di codice all'interno di un template. Si utilizza per implementare un comportamento simile a un'istruzione switch-case in altri linguaggi di programmazione, dove diverse condizioni vengono verificate e il blocco di codice associato al caso corrispondente viene eseguito.

Example

```
<div [ngSwitch]="heroSelected.name">
  <div *ngSwitchCase="'Iron Man'">
    <h3>Iron Man</h3>
    <p>Iron Man è un genio miliardario, playboy e filantropo che combatte per la giustizia!</p>
  </div>
  <div *ngSwitchCase="'Spider-Man'">
    <h3>Spider-Man</h3>
    <p>Spider-Man è un giovane eroe che utilizza i suoi poteri per proteggere New York!</p>
  </div>
  <div *ngSwitchDefault>
    <h3>Eroe Non Selezionato</h3>
    <p>Seleziona un eroe dalla lista per visualizzare ulteriori dettagli!</p>
  </div>
</div>
```

In questo esempio:

- `ngSwitch` è la direttiva che viene applicata all'elemento contenitore e associa un'espressione da valutare.
- `ngSwitchCase` viene applicato agli elementi figlio del contenitore e rappresenta un singolo caso da confrontare con il valore dell'espressione.

***ngFor**

La direttiva `*ngFor` è una direttiva strutturale di Angular che permette di iterare su una collezione di dati, come un array o una lista, e generare dinamicamente un elemento del DOM per ogni elemento della collezione.

Example

```
<ul>
  <li *ngFor="let hero of heroes">{{ hero.name }}</li>
</ul>
```

In questo esempio:

- `let hero` è la variabile che rappresenta ciascun elemento dell'array durante l'iterazione.
- `heroes` è la collezione di dati che si desidera iterare.

In Angular, puoi utilizzare l'attributo index all'interno della direttiva `*ngFor` per ottenere l'indice dell'elemento corrente nell'iterazione. L'indice è numerato a partire da 0 per il primo elemento.

Example

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

In questo esempio:

- `let i = index` definisce una variabile i che rappresenta l'indice dell'elemento corrente.
- `{{ i }}` stampa l'indice dell'elemento nell'array, tramite interpolazione (lo vedremmo nella sezione successiva dedicata al data binding).

Tip

La direttiva `trackBy` in Angular è una funzionalità avanzata che viene utilizzata insieme a `*ngFor` per migliorare le prestazioni quando si lavora con liste dinamiche. `trackBy` permette ad Angular di identificare univocamente gli elementi all'interno di una lista, in modo che l'engine di rendering possa determinare con maggiore precisione quali elementi sono cambiati, aggiunti o rimossi, evitando di rifare il rendering dell'intera lista ogni volta che cambia uno degli elementi, ma solo degli elementi che sono cambiati.

Sintassi:

```
<li *ngFor="let hero of heroes; trackBy: trackByHeroId">
  {{ item.name }}
</li>
```

`trackBy: trackByHeroId` specifica una funzione personalizzata che Angular utilizza per identificare in modo univoco ogni elemento nella lista. Questa funzione deve restituire un valore univoco per ogni elemento.

```
trackByHeroId(index: number, hero: Hero): number {
    return hero.id;
}
```

`trackByHeroId` è una funzione che restituisce `hero.id`, un valore univoco per ogni eroe. Questo permette ad Angular di tracciare ogni elemento della lista in modo efficiente.

ngClass

`ngClass` permette di aggiungere, rimuovere o cambiare classi CSS su un elemento HTML in modo dinamico. È molto utile quando vogliamo applicare stili diversi in base a condizioni specifiche senza dover scrivere logica complicata nel template.

Example

```
<div [ngClass]="{'active': isActive, 'disabled': !isActive}">
  Pulsante
</div>
```

Se `isActive` è true, il div riceverà la classe active; altrimenti riceverà la classe disabled. ngClass può accettare anche array di classi o semplici stringhe di classi separate da spazi.

ngStyle

`ngStyle` permette di applicare stili CSS direttamente sull'elemento HTML in maniera dinamica, senza dover passare attraverso le classi.

Example

```
<div [ngStyle]="{'color': isActive ? 'green' : 'red', 'font-size': '20px'}">
  Testo dinamico
</div>
```

Il colore del testo sarà verde se `isActive` è true, oppure rosso se `isActive` è false, e avrà sempre una dimensione del font di 20px.

4.2.2 Data Binding

Il data binding è il meccanismo che collega i dati del componente alla vista (template HTML) e viceversa. Oltre a gestire l'interazione tra logica e interfaccia, il data binding consente anche il passaggio di dati tra componenti lungo la loro gerarchia, facilitando così la comunicazione tra componenti padre e figlio.

Grazie al data binding, è possibile aggiornare dinamicamente la visualizzazione dei dati nella pagina e gestire in modo reattivo gli eventi dell'utente, come click, input o cambiamenti di stato.

Angular offre quattro principali modalità di data binding:

Tipo	Descrizione	Sintassi
Interpolazione	Collega dati dal componente alla vista, per visualizzare valori.	`{{ proprietà }}`
Property Binding	Collega dati dal componente ad attributi/proprietà HTML.	`[attributo]="proprietà"`
Event Binding	Collega eventi della vista al metodo del componente.	`(evento)="metodo()"`
Two-way Binding	Sincronizza dati in entrambe le direzioni tra componente e vista.	`[(ngModel)]="proprietà"`

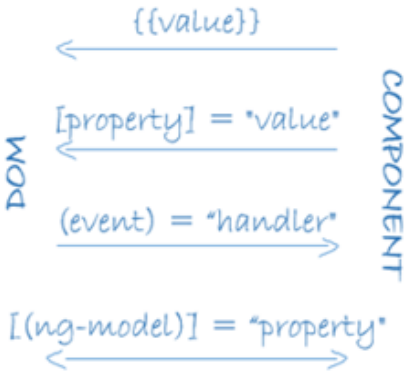

Figura 5: Schema tipologie di data binding

Interpolazione ({{ }})

Permette di inserire valori direttamente nell'HTML

Example

```
export interface Hero {
  name: string;
  surname: string;
  isSelected: boolean;
  imageUrl: string;
  creationDate: string;
}

...

export class HelloWorldComponent {

  public user: Hero;

  constructor(){
    this.user = {name: Mario,
      surname: Rossi,
      imageUrl:'./assets/logo.png',
    };
  }

}
```

Nel template:

Example

```
<p>Benvenuto, {{ user.name }}!</p>
```

Nell'esempio grazie alla interpolazione è possibile visualizzare valori salvati in una variabile o in un campo di oggetto all'interno dell'HTML.

Property Binding ([ ])

Collega una proprietà del componente a una proprietà HTML o di un componente figlio.

Riprendendo il ts precedente:

Example

```
<img [src]="user.imageUrl" alt="{{ user.name }}">
```

Event Binding (( ))

Permette di rispondere agli eventi dell'utente, come click, input, submit, ecc.

Example

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

Nel componente:

Example

```
public selectHero(hero: Hero):void{
  this.selectedHero = hero;
}
```

Quando l'utente clicca il pulsante, verrà eseguito il `selectHero` passando l'elemento cliccato.

Two-way Binding ([( )])

Combina property binding ed event binding: sincronizza il valore sia dal componente alla vista, sia dalla vista al componente.

Example

```
<input [(ngModel)]="hero.nome">
<p>Hai scritto: {{ hero.nome }}</p>
```

Grazie alla direttiva `ngModel` un valore di proprietà dati scorre nella casella di input dal componente come con l'associazione di proprietà. Le modifiche dell'utente tornano anche al componente, reimpostando la proprietà sul valore più recente, come con l'associazione di eventi.

Inoltre grazie al Two-way Binding è pssibile la comunicazione tra componenti padre e figlio.

Nel template padre:

Example

```
<app-hero-detail [hero]="hero" (heroSelected)="onHeroSelectedDetails($event)"></app-hero-detail>
```

Nel component figlio:

Example

```
...
export class HeroDetailComponent {
  @Input() hero: Hero = '';
  @Output() heroSelected: EventEmitter<string> = new EventEmitter<string>();

  onDetails(details: string) {
    this.heroSelected.emit(details);
  }
}
```

Tramite la sintassi `[hero]="hero"`, il padre passa al figlio un oggetto Hero, sfruttando il meccanismo del property binding di Angular. Il valore assegnato all'input del figlio (`@Input() hero`) viene quindi popolato direttamente con l'oggetto fornito dal padre.

Allo stesso tempo, il padre si mette in ascolto dell'evento personalizzato (`heroSelected`). Questo evento viene definito nel figlio usando `@Output()` e un `EventEmitter<string>`, che consente al figlio di notificare al padre un'azione o un cambiamento. Quando il figlio esegue il metodo `onDetails(details: string)`, chiama `this.heroSelected.emit(details)`, emettendo così l'evento verso il padre. Il padre intercetta l'evento attraverso la sintassi `(heroSelected)="onHeroSelected($event)"`, richiamando il proprio metodo onHeroSelected e passando come parametro il valore emesso dal figlio.

Ciò che viene emesso dal component figlio è ricavabile dal padre attraverso la variabile `$event`.

4.3 Ciclo di vita di un componente (Lifecycle)

Gli Angular Lifecycle Hooks (metodi del ciclo di vita) permettono di eseguire codice in momenti specifici della vita di un componente o di una direttiva. Ogni hook corrisponde a una fase precisa: dall'inizializzazione alla distruzione.

Hook	Descrizione
`ngOnChanges()`	Viene eseguito ogni volta che cambia il valore di una proprietà di input. Si attiva prima dell'inizializzazione del componente e ogni volta che uno degli input viene modificato.
`ngOnInit()`	Rappresenta la fase di inizializzazione del componente. Viene eseguito una sola volta dopo il primo `ngOnChanges()`.
`ngDoCheck()`	Metodo richiamato durante il controllo interno di Angular per intercettare modifiche ai dati o allo stato dei componenti. Segue ogni esecuzione di `ngOnChanges()` e `ngOnInit()`.
`ngAfterContentInit()`	Eseguito dopo che Angular ha proiettato il contenuto nel componente. Viene chiamato una sola volta, immediatamente dopo il primo `ngDoCheck()`.
`ngAfterContentChecked()`	Richiamato dopo ogni verifica del contenuto proiettato. Si attiva dopo `ngAfterContentInit()` e successivamente a ogni `ngDoCheck()`.
`ngAfterViewInit()`	Indica che la vista e i componenti figli sono stati inizializzati. È eseguito una sola volta dopo `ngAfterContentChecked()`.
`ngAfterViewChecked()`	Si verifica dopo ogni verifica della vista del componente. Chiamato dopo `ngAfterViewInit()` e successivamente a ogni `ngAfterContentChecked()`.
`ngOnDestroy()`	Ultima fase del ciclo di vita: chiamato immediatamente prima che Angular distrugga il componente. Serve per liberare risorse, annullare sottoscrizioni e rimuovere event handler per evitare memory leaks.

4.4 ng-template

`ng-template` è un contenitore invisibile per contenuto che Angular non renderizza immediatamente nel DOM. Il contenuto all'interno di un ng-template viene creato e inserito solo quando serve, ad esempio in base a condizioni o strutture dinamiche

Example

```
<div *ngIf="customerData; else loading">

<ng-template #loading>
  <p>Caricamento in corso...</p>
</ng-template>
```

Il paragrafo sarà creato solo se customerData è false, senza aggiungere elementi extra nel DOM. Viene creato un riferimento nel `ng-template` (`#loading`) per indicare a che condizione deve riferirsi.

4.5 ng-content

`ng-content` è usato per realizzare il content projection, ovvero per permettere di inserire contenuto dinamico dentro un componente da parte del suo genitore. Il componente definisce un punto (`<ng-content>`) in cui chi lo usa può "iniettare" del contenuto HTML.

Grazie a `ng-content` possiamo rendere i componenti riutilizzabili e flessibili, lasciando scegliere all'esterno cosa mostrare dentro.

Esempio:

Nel componente figlio app-card:

Example

```
<div class="card">
  <ng-content></ng-content>
</div>
```

Nel componente padre:

Example

```
<app-card>
  <p>Questo paragrafo sarà proiettato dentro la card!</p>
</app-card>
```

Il tag `<p>` viene automaticamente inserito dove si trova `<ng-content>` nel figlio.

4.6 ng-container

`ng-container` è un contenitore logico che permette di raggruppare più elementi senza creare nuovi nodi nel DOM. Serve soprattutto quando vogliamo applicare direttive strutturali come `*ngIf` , `*ngFor` , `ngSwitch` su più elementi contemporaneamente, senza introdurre tag superflui come `<div>` .

`ng-container` non si vede nel DOM: il browser non renderizza nulla per esso.

Example

```
<ng-container *ngIf="isLoggedIn">
  <p>Benvenuto!</p>
  <button>Logout</button>
</ng-container>
```

Il paragrafo e il bottone saranno creati solo se `isLoggedIn` è vero, senza aggiungere elementi extra nel DOM.

5. Pipes

In Angular, le pipe sono speciali operatori che consentono di trasformare i dati direttamente nel template, in modo semplice e dichiarativo. Sono molto utili per formattare dati come date, numeri, stringhe, valute, percentuali e altro, senza dover scrivere codice extra nei componenti.
La pipe è una classe con il decoratore @Pipe in cui viene definita una funzione che trasforma valori in input in valori in output da visualizzare in una vista. Angular offre un set di pipe built-in attreverso il package @angular/common:

Nome	Descrizione
AsyncPipe	Legge il valore da una Promise o da un Observable di RxJS.
CurrencyPipe	Trasforma un numero in una stringa di valuta, formattata secondo le regole locali.
DatePipe	Formatta un valore di tipo Date secondo le regole locali.
DecimalPipe	Trasforma un numero in una stringa con punto decimale, formattata secondo le regole locali.
I18nPluralPipe	Mappa un valore a una stringa che lo pluralizza secondo le regole locali.
I18nSelectPipe	Mappa una chiave a un selettore personalizzato che restituisce un valore desiderato.
JsonPipe	Trasforma un oggetto in una rappresentazione stringa tramite `JSON.stringify`, utile per il debug.
KeyValuePipe	Trasforma un oggetto o una mappa in un array di coppie chiave-valore.
LowerCasePipe	Trasforma il testo in minuscolo.
PercentPipe	Trasforma un numero in una stringa percentuale, formattata secondo le regole locali.
SlicePipe	Crea un nuovo array o una nuova stringa contenente una porzione (slice) degli elementi.
TitleCasePipe	Trasforma il testo in formato titolo (maiuscola all'inizio di ogni parola).
UpperCasePipe	Trasforma il testo in maiuscolo.

5.1 Utilizzo

L'operatore pipe di Angular utilizza il carattere della barra verticale (|) all'interno di un'espressione nel template. È un operatore binario: l'operando a sinistra rappresenta il valore da trasformare, mentre l'operando a destra indica il nome della pipe e gli eventuali parametri aggiuntivi.

Example

```
<p>Totale: {{ amount | currency }}</p>
```

Per specificare un parametro aggiuntivo, basta inserire due punti (:) dopo il nome della pipe, seguiti dal valore del parametro.

Example

```
<p>L'evento si terrà alle {{ scheduledOn | date:'hh:mm' }}.</p>
```

Nel caso di più parametri

Example

```
<p>L'evento si terrà alle {{ scheduledOn | date:'hh:mm':'UTC' }}.</p>
```

5.2 Pipes Custom

In Angular è possibile definire delle pipes custom.

Example

```
@Pipe({ name: 'reverse' })
export class ReversePipe implements PipeTransform {
    transform(value: string): string {
        return value.split('').reverse().join('');
    }
}
```

La pipe ha il nome 'reverse', quindi potrà essere utilizzata nei template con la sintassi {{ valore | reverse }}.
La classe ReversePipe implementa l'interfaccia PipeTransform, il che significa che deve definire un metodo transform. Questo metodo riceve un valore in input, in questo caso una stringa, e restituisce una nuova stringa con i caratteri invertiti.

Opzionalmente possono essere aggiunto parametri aggiuntivi indicandoli in input nel metodo trasform.

5.3 Pipes Pure

Per ottimizzare le performance, Angular mette a disposizione le pipe pure. Le pipe pure in Angular sono pipe che vengono eseguite solo quando cambia l'input di riferimento, ossia quando cambia effettivamente il valore passato alla pipe (o uno dei suoi argomenti).

Per indicare una pipe come pure basta aggiungere nel decoratore @Pipe, true nel campo pure.

Example

```
@Pipe({
    name: 'reverse',
    pure: true,
})
```

6. Services

In Angular, un servizio è una classe che contiene logica riutilizzabile e funzionalità condivise tra più componenti o moduli. Permette di organizzare il codice, separando le responsabilità: ad esempio, anziché scrivere la logica di accesso ai dati direttamente in un componente, si crea un servizio dedicato, e il componente lo utilizza quando necessario, rendendo così l'applicazione più flessibile inserendo un provider dedicato all'accesso dati.

Un servizio può occuparsi di per esempio: chiamate HTTP a un'API, gestione dello stato dell'applicazione, operazioni matematiche, accesso a dati locali, formattazioni, e così via. Un servizio è una classe TypeScript annotata con il decoratore `@Injectable()`.

6.1 Dependency Injection

Per usare un servizio in un componente, Angular usa un meccanismo chiamato Dependency Injection.

La Dependency Injection è un meccanismo che fornisce automaticamente le dipendenze richieste da una classe, senza che sia la classe stessa a crearle. Quindi un oggetto (consumer) delega il compito di fornire le dipendenze necessarie a del codice esterno (injector). In Angular, le dipendenze sono in genere servizi, ma possono essere anche altri oggetti o classi.

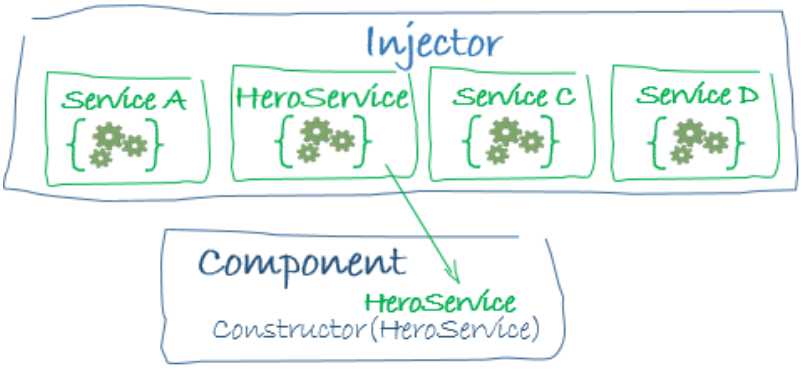

Figura 6: Schema funzionamento DI

Se un componente ha bisogno di un servizio, Angular lo "inietta" nel costruttore del componente:

Example

```
@Component({...})
export class ProfiloComponent implements OnInit {
    constructor(private datiUtenteService: DatiUtenteService) {}

    ngOnInit(): void {
        const nome = this.datiUtenteService.getNomeUtente();
    }
}
```

In questo esempio, Angular crea un'istanza di `DatiUtenteService` e la passa al costruttore di `ProfiloComponent` automaticamente. Il componente non si preoccupa di come viene creato il servizio, si limita a usarlo. Perciò, un componente che vuole utilizzare il servizio inserisce la sua dichiarazione nel costruttore. E' fondamentale annotare il parametro con il tipo corretto che sarà la chiave usata dall'injector per capire che istanza iniettare.

6.2 Injectables

Come scritto precedentemente, i servizi possono essere iniettati in un componente come dipendenze, e un servizio è una classe che utilizza il decoratore `@Injectable`. Inoltre un servizio puo avere sua volta delle dipendenze.

Nel seguente esempio viene mostrato come creare un servizio che a sua volta ha una dipendenza, risolta tramite l'injection.

Example

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

@Injectable({
providedIn: 'root'
})
export class UtenteService {
    constructor(private http: HttpClient) {}

    getUtenti() {
        return this.http.get('/api/utenti');
    }
}
```

Nell'esempio `@Injectable` contiene dei metadati. Infatti, la proprietà `providedIn` indica a Angular dove registrare il servizio nel sistema di iniezione delle dipendenze. Di seguito i valori che puo assumere: - 'root': È l'opzione più comune e consigliata. Significa che il servizio sarà disponibile ovunque nell'applicazione, quindi creato una sola volta (singleton) e sarà caricato automaticamente da Angular, senza doverlo registrare manualmente nel modulo. - 'any': Questa opzione crea istanze separate del servizio in ogni modulo che lo richiede, utile per servizi che non devono condividere stato tra moduli. - 'SomeModule' : permette di indicare un modulo specifico in cui fornire il servizio. Questo è utile quando vuoi limitare l'ambito del servizio a un modulo particolare (come un modulo lazy-loaded). Nel providedIn deve essere indicato il nome del modulo, per esempio: `providedIn: MyFeatureModule`

6.3 Providers

Nel capitolo dedicato ai componenti, abbiamo visto che il decoratore `@Component` include una proprietà chiamata `providers`. In Angular, i providers rappresentano il meccanismo attraverso cui viene definito come fornire i servizi (dependencies) all'interno dell'applicazione, utilizzando il sistema di Dependency Injection. Quando un servizio viene registrato come provider, si comunica ad Angular come creare o recuperare un'istanza di quel servizio. Sarà poi compito dell'Injector mantenere e gestire la mappa tra i token (le chiavi) e le istanze associate.

Abbiamo già incontrato la registrazione tramite `@Injectable({ providedIn: 'root' })`, che rende il servizio un singleton a livello globale, condiviso da tutta l'applicazione. Tuttavia, è anche possibile registrare un servizio direttamente all'interno della proprietà providers di un componente: in questo modo il servizio è scoped al componente stesso, ovvero vive solo finché vive il componente, e ogni istanza del componente riceve una propria istanza del servizio.

Example

```
@Component({
    ....
    providers: [LoggerService]  // provider locale al componente
    })
    export class ChildComponent {
    constructor(private logger: LoggerService) {}

    logMessage() {
        this.logger.log('Messaggio dal ChildComponent');
    }

}
```

In questo esempio, Angular crea un'istanza locale di `LoggerService` per ogni istanza del componente. L'istanza viene creata tramite il costruttore della classe, sfruttando implicitamente la strategia `useClass`, in cui si specifica solo la chiave (`LoggerService`) e Angular deduce la classe da instanziare.

Example

```
@Component({
    ....
    providers: [{ provide: LoggerService, useClass: LoggerService }]
    })
```

La configurazione estesa di un provider è un oggetto letterale con due proprietà: - La proprietà provide contiene il token, che funge da chiave per richiedere il valore della dipendenza. - La seconda proprietà è un oggetto di definizione del provider, che indica all'injector come creare il valore della dipendenza. La definizione del provider sarà tratta nelle prossime sezioni.

6.3.1 useClass

Permette di fornire una classe alternativa quando viene richiesto un certo token. useClass consente di creare e restituire una nuova istanza della classe specificata.

Example

```
@Component({
    ....
    providers: [{ provide: MyService, useClass: MockService }]
    })
```

In questo caso, qualsiasi classe che richieda `MyService` riceverà in realtà un'istanza di `MockService`.

Per rendere il servizio disponibile in tutta l'applicazione, lo stesso provider può essere registrato direttamente nella definizione del modulo root. In questo modo sarà usata una sola istanza di `MockService` per tutti i componenti del modulo.

6.3.2 useExisting

useExisting consente di collegare un token a un altro, creando un alias. In questo modo, il primo token diventa un riferimento alternativo al servizio associato al secondo token, permettendo di accedere allo stesso oggetto tramite due nomi diversi.

Example

```
@Component({
    ....
    providers: [ NewLogger, { provide: OldLogger, useExisting: NewLogger }]
    })
```

Nell'esempio precedente, l'injector fornisce la stessa istanza singleton di `NewLogger` quando un componente richiede sia `NewLogger` sia `OldLogger`. In questo modo, `OldLogger` è un alias di `NewLogger`.

Warning

Non utilizzare `useClass` per creare un alias tra `OldLogger` e `NewLogger`, perché questo comporterebbe la creazione di due istanze distinte di `NewLogger`.

6.3.3 useFactory

useFactory permette di generare un oggetto dipendenza attraverso una funzione factory. Questo approccio è ideale quando è necessario creare valori dinamici, basati su dati disponibili nel sistema di Dependency Injection o provenienti da altre parti dell'applicazione.

Nell'esempio seguente, si crea un servizio `ApiService` che utilizza un valore di configurazione dinamico (apiUrl) fornito da un `ConfigService`.

Example

```
//config.service.ts
import { Injectable } from '@angular/core';

@Injectable()
export class ConfigService {
    private environment = 'production';
    private configs = {
        development: { apiUrl: 'http://localhost:3000' },
        production: { apiUrl: 'https://api.example.com' }
    };

    get apiUrl(): string {
        return this.configs[this.environment].apiUrl;
    }
}
```

```
//api.service.ts
export class ApiService {
    constructor(private apiUrl: string) {}

    getEndpoint(): string {
        return `${this.apiUrl}/endpoint`;
    }
}
```

In base all'ambiente viene fornito un url differente da `ConfigService`. Per utilizzare il useFactory:

Example

```
//api.service.provider.ts
import { ApiService } from './api.service';
import { ConfigService } from './config.service';

export function apiServiceFactory(config: ConfigService): ApiService {
    return new ApiService(config.apiUrl);
}

export const apiServiceProvider = {
    provide: ApiService,
    useFactory: apiServiceFactory,
    deps: [ConfigService]
};
```

Il campo useFactory specifica che il provider è una funzione factory implementata da `apiServiceFactory`. La funzione factory `apiServiceFactory` accede a `ApiService`. La proprietà deps è un array di token di provider. Angular risolve questi token e inietta i servizi corrispondenti nei parametri della funzione factory, nell'ordine specificato.

Example

```
@Component({
    ...
    providers: [ConfigService, apiServiceProvider]
})
export class AppComponent {
 endpoint: string;

    constructor(private apiService: ApiService) {
        this.endpoint = this.apiService.getEndpoint();
    }
}
```

Nel component dove viene utilizzato `ApiService`, vengono indicati nel providers il provider factory `apiServiceProvider` e la sua dipendenza `ConfigService`.

6.3.4 useValue

useValue consente di associare un valore statico a un token di Dependency Injection. Questa tecnica è particolarmente utile per fornire costanti di configurazione a runtime, come URL di base, flag di funzionalità o valori predefiniti. Inoltre, è molto usata nei test unitari per sostituire servizi reali con dati mock, semplificando la scrittura e il controllo dei test. Per utilizzare useValue occore usare l'oggetto `InjectionToken` come token del provider per le dipendenze che non sono classi.

Example

```
//app.config.ts
import { InjectionToken } from '@angular/core';

export interface AppConfig {
    title: string;
}

export const APP_CONFIG = new InjectionToken<AppConfig>('Descrizione del token app.config');
```

Nell'esempio viene definito un token `APP_CONFIG` di tipo `InjectionToken`. Il parametro di tipo opzionale `<AppConfig>` e la descrizione 'Descrizione del token app.config' servono a specificare lo scopo del token.

Example

```
//app.component.ts
const MY_APP_CONFIG_VARIABLE: AppConfig = {
    title: 'Ciao',
};

providers: [{ provide: APP_CONFIG, useValue: MY_APP_CONFIG_VARIABLE }]
```

Viene registrato il provider della dipendenza nel componente usando l'oggetto `InjectionToken` `APP_CONFIG`.

Example

```
// app.component.ts
const MY_APP_CONFIG_VARIABLE: AppConfig = {
    title: 'Ciao',
};

@Component({
    selector: 'app-root',
    template: `...`,
    providers: [{ provide: APP_CONFIG, useValue: Y_APP_CONFIG_VARIABLE }]
})
export class AppComponent {
    apiUrl: string;

    constructor() {
        const config = inject(APP_CONFIG);
        this.apiUrl = config.apiUrl;
    }
}
```

Infine la configurazione del token viene inietta nel costruttore tramite la funzione `Inject`.

6.4 Injector

In Angular, l'Injector è il meccanismo centrale che gestisce la creazione e risoluzione delle dipendenze, mantenendo una mappa di provider (token). Normalmente non lo si usa direttamente, perché Angular gestisce automaticamente la Dependency Injection tramite i costruttori. Tuttavia, è possibile fare iniezione manuale usando l'Injector, ad esempio in contesti dinamici o avanzati.

Nel seguente esempio viene illustrata un iniezione manuale con `Inject`.

Example

```
@Component({
selector: 'app-example',
template: `...`,
providers: [LoggerService]
})
    export class ExampleComponent {
    constructor(private injector: Injector) {
        const logger = this.injector.get(LoggerService);
    }
}
```

Tramite il metodo `get()` recupera l'istanza del servizio da iniettare.

7. Routing

In Angular, le routes (rotte) sono il meccanismo che consente di gestire la navigazione tra viste (routing) in una Single Page Application (SPA). Il modulo che gestisce questa funzionalità è il `RouterModule`.

7.1 Impostare il routing

Per impostare il routing occore specificare qual'è l'URL di base da cui il router può comporre gli URL per tutte le view. L'URL di base è impostato nel file `index.html` con il tag `<base>` (`<base href="/">`) all'interno del tag `<head>`

Si specificano poi nel modulo le rotte attraverso l'oggetto Routes

Example

```
// app-routing.module.ts
const routes: Routes = [
    { path: '', component: HomeComponent },             // rotta base
    { path: 'about', component: AboutComponent },       // rotta /about
    { path : '** ', component : PageNotFoundComponent } // rotta wildcard,
];
```

Un oggetto Routes è un array di oggetti Routes che hanno se seguenti proprietà:

Proprietà	Tipo	Descrizione
`path`	`string`	Il percorso dell'URL che attiva la rotta (es. `'home'`, `'user/:id'`).
`component`	`Type<any>`	Il componente da visualizzare quando il percorso viene attivato.
`redirectTo`	`string`	Percorso verso cui reindirizzare in caso di corrispondenza.
`pathMatch`	`'full'` | `'prefix'`	Specifica come confrontare il path: `'full'` per corrispondenza completa, `'prefix'` per parziale (default).
`children`	`Route[]`	Array di rotte figlie per creare una struttura gerarchica (nested routes).
`loadChildren`	`string` | `() => ...`	Permette il lazy loading di un modulo figlio.
`canActivate`	`any[]`	Array di guardie che controllano se la rotta può essere attivata.
`canActivateChild`	`any[]`	Guardie applicate ai figli della rotta.
`canDeactivate`	`any[]`	Guardie che decidono se è possibile uscire dalla rotta corrente.
`canLoad`	`any[]`	Guardie che decidono se un modulo può essere caricato (usato con `loadChildren`).
`resolve`	`{ [key: string]: any }`	Oggetto che associa chiavi a resolver per caricare dati prima di attivare la rotta.
`data`	`{ [key: string]: any }`	Oggetto contenente dati statici (es. titolo, meta, flag) associati alla rotta.
`runGuardsAndResolvers`	`'paramsChange' | 'pathParamsChange' | 'always'`	Specifica quando ricaricare guardie e resolver se la rotta è già attiva. (solo per Angular 12)
`outlet`	`string`	Nome dell'outlet dove caricare il componente (default: `'primary'`).

Note

L'ordine delle rotte è importante perché il Router utilizza una strategia "prima corrispondenza vince" quando effettua il matching delle rotte. Per questo motivo, le rotte più specifiche devono essere elencate prima di quelle meno specifiche.

Elenca quindi per prime: - le rotte con un percorso statico, - poi una rotta con percorso vuoto (''), che rappresenta la rotta di default, - e infine la wildcard route ('**'), che corrisponde a qualsiasi URL e viene selezionata solo se nessun'altra rotta corrisponde prima.

Una volta definite le rotte è necessario richiamare il metodo `forRoot()` (spiegato nel capitolo dedicato ai moduli).

Example

```
// app-routing.module.ts

const routes: Routes = [
    { path: '', component: HomeComponent },
    { path: 'about', component: AboutComponent },
    { path : '** ', component : PageNotFoundComponent }
];

@NgModule({
    imports: [RouterModule.forRoot(routes)],
    exports: [RouterModule]
})
export class AppRoutingModule {}
```

Il modulo è configurato con le route ed esportato agli altri moduli dell'applicazione (Ad esempio nell'AppModule). Se si vuole definire delle sotto rotte in un altro modulo si dovrà utilizzare il metodo `forChild()`. Infine nel file `index.html`, inserire la direttiva `<router-outlet>`. L'elemento specifica in quale posto del DOM vanno inserite le varie view gestite dal router.

7.2 RouterLink

Un modo semplice per abilitare la navigazione tra le pagine in un'applicazione Angular è l'utilizzo della direttiva `RouterLink`. Quando applicata a un elemento del DOM, `RouterLink` trasforma l'elemento in un collegamento che, se cliccato, attiva la navigazione verso una specifica route definita nel router. La navigazione risultante visualizzerà uno o più componenti instradati all'interno di uno o più elementi `<router-outlet>` presenti nella pagina.

Example

```
<li class="nav-item">
    <a
        class="nav-link"
        routerLink="/tariffe"
        routerLinkActive="active">
        Tariffe
    </a>
</li>
```

7.3 Path params

In Angular, per passara i path params basta aggiungere nella rotta ":" seguito dal nome del parametro.

Example

```
// app-routing.module.ts
...
const routes: Routes = [
    { path: '', component: HomeComponent },
    { path: 'about', component: AboutComponent },
    { path: 'user/:id', component: UserComponent },
    { path : '** ', component : PageNotFoundComponent }
];
...
```

Tramite `routerLink` bastera indicare il nome della rotta e il valore da comunicare: `<a [routerLink]="['/user', 42]">Vai a User 42</a>`, oppure tramite typescript `this.router.navigate(['/user', 42]);`.

per leggere i valori nel component occore utilizzare il service `ActivatedRoute`:

Example

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

constructor(private route: ActivatedRoute) {}

ngOnInit() {
    this.route.paramMap.subscribe(params => {
        const id = params.get('id');
    });
}
```

7.4 Nesting routes

Le rotte annidate (nesting routes) in Angular permettono di organizzare l'applicazione inserendo componenti figli all'interno di un componente genitore, sfruttando il sistema di routing. Quando l'app cresce in complessità, può essere utile definire rotte relative a componenti diversi da quello radice, per strutturare meglio la navigazione e la visualizzazione dei contenuti.

In questi casi, viene introdotto un secondo `<router-outlet>` all'interno del componente genitore, in aggiunta a quello già presente nell' `AppComponent`.

Ad esempio, supponiamo di avere due componenti figli: `child-a` e `child-b`. Il componente `FirstComponent`, che funge da contenitore, include un proprio `<nav>` e un secondo `<router-outlet>`, nel quale verranno renderizzati dinamicamente i componenti figli in base alla rotta attiva.

Example

```
<h2>First Component</h2>
<nav>
    <ul>
        <li><a routerLink="child-a">Child A</a></li>
        <li><a routerLink="child-b">Child B</a></li>
    </ul>
</nav>
<router-outlet />
```

Una rotta figlia è come qualsiasi altra rotta: ha bisogno sia di un percorso (path) che di un componente. L'unica differenza è che le rotte figlie vengono definite all'interno di un array children all'interno della rotta padre.

Example

```
const routes: Routes = [
    {
        path: 'first-component',
        component: FirstComponent,
        children: [
            {
                path: 'child-a',
                component: ChildAComponent,
            },
            {
                path: 'child-b',
                component: ChildBComponent,
            },
        ],
    },
];
```

7.5 Guardie

Le guardie in Angular sono dei meccanismi di protezione che permettono di controllare l'accesso alle rotte della tua applicazione. Le guardie eseguono controlli prima di attivare, disattivare, caricare o uscire da una rotta.

Di seguito viene riportato un esempio di guardia per controllare se lo user ha effettuato il login e quindi accedere alla dashboard:

Example

```
@Injectable({
    providedIn: 'root'
})
export class AuthGuard implements CanActivate {
    constructor(private authService: AuthService, private router: Router) {}

    canActivate(): boolean {
        if (this.authService.isLoggedIn()) {
            return true;
        }
        this.router.navigate(['/login']);
        return false;
    }
}
```

Viene implementata l'interfaccia `CanActivate` definendo il metodo `canActivate`.

Example

```
const routes: Routes = [
    { path: 'dashboard', component: DashboardComponent, canActivate: [AuthGuard] },
    { path: 'login', component: LoginComponent }
];
```

Utilizzando la proprietà dell'oggetto Route dedicata alle guardie più idonea (vedi tabella dedicata all'oggetto Route), viene aggiunta la guardia `AuthGuard`.

7.6 Lazy loading

In Angular, è possibile configurare le rotte per caricare i moduli/componenti in modo lazy (caricamento a richiesta), il che significa che Angular carica i moduli/componenti solo quando necessario, invece di caricarli tutti all'avvio dell'applicazione.

Ogni rotta può caricare lazy il proprio modulo/componente stand-alone utilizzando loadComponent:

Example

```
const routes: Routes = [
    {
        path: 'lazy',
        loadComponent: () => import('./lazy.component').then(c => c.LazyComponent)
    }
];
```

Con `loadComponent: () => import('./lazy.component')...` viene usato il lazy loading di un componente stand-alone. Angular non carica immediatamente `LazyComponent` all'avvio dell'applicazione, ma solo quando l'utente naviga a /lazy. Con `.then(c => c.LazyComponent)`, dopo l'importazione dinamica del file `lazy.component.ts`, si estrae il `LazyComponent` dal modulo importato.

8. Observables

Un Observable è un oggetto che rappresenta un flusso di dati nel tempo, utile per gestire eventi asincroni. Si basa sul pattern **Observer** e viene utilizzato per scambiare messaggi tra due entità: il **publisher** (emettitore) e il **subscriber** (consumatore). In Angular, tutto questo è implementato tramite la libreria RxJS. Gli Observable sono fondamentali per la gestione reattiva e asincrona dei dati, come richieste HTTP, eventi dell'utente o dati che cambiano dinamicamente nel tempo. Un Observable definisce una funzione che emette valori a uno o più observer. Tuttavia, questa funzione non viene eseguita finché un subscriber non si iscrive utilizzando il metodo `subscribe()`.

Una volta iscritta, l'entità riceve notifiche tramite tre metodi principali:

- `next(value)`: emette un nuovo valore al subscriber;
- `error(err)`: segnala un errore e termina il flusso (nessun altro valore verrà emesso dopo);
- `complete()`: indica che il flusso di dati è terminato con successo. Questi tre metodi fungono da handler degli eventi ricevuti.

La ricezione dei dati continua fino a che:

- l'Observable completa (ovvero ha finito di emettere valori e notifica ai subscriber la fine del flusso),
- si verifica un errore,
- oppure il subscriber si disiscrive manualmente tramite `unsubscribe()`.

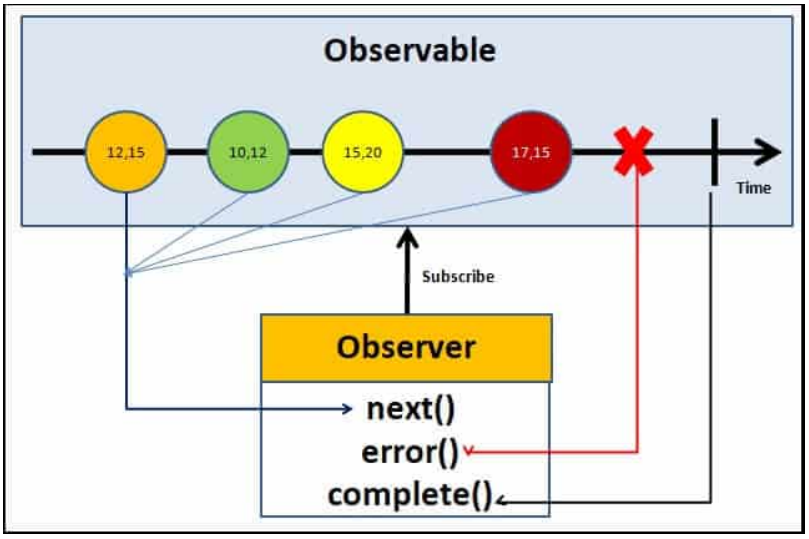

Figura 7: Schema Observable

Example

```
// Creazione di un Observable
import { Observable } from 'rxjs';

const esempio$ = new Observable<string>((observer) => {
    observer.next('Primo valore');
    observer.next('Secondo valore');

    // Simula un errore
    //observer.error('Si è verificato un errore');

    // Completa il flusso
    observer.complete();
});

// Sottoscrizione all'Observable
esempio$.subscribe({
    next: (valore) => console.log('Ricevuto:', valore), // next: può essere omesso
    error: (err) => console.error('Errore:', err),
    complete: () => console.log('Flusso completato')
});
```

Se si scommenta `observer.error()`, il flusso termina con un errore e `complete()` non viene mai chiamato.

8.1 RxJS

Come già anticipato, Reactive Extensions for JavaScript (RxJS) è una libreria utilizzata da Angular per semplificare lo sviluppo di codice asincrono e basato su eventi, sfruttando la potenza degli Observable.

RxJS fornisce:

- l'implementazione della classe `Observable`,
- una serie di operatori per trasformare, filtrare o combinare flussi di dati,
- utility per creare Observable a partire da altri tipi di dati, come array, promise, eventi del DOM o timer.

Di seguito vengono illustrati alcuni operatori e creator RxJS in Angular 12:

Nome	Tipo	Categoria	Input accettato	Descrizione
`of`	Creator	Creation	Valori statici	Crea un Observable che emette uno o più valori statici.
`from`	Creator	Creation	Array, Promise, Iterable	Converte un array, promise o iterable in un Observable.
`fromEvent`	Creator	Creation	Eventi DOM	Crea un Observable da un evento (es. click, input).
`map`	Operatore	Transformation	Observable	Trasforma ogni valore emesso.
`filter`	Operatore	Filtering	Observable	Emette solo i valori che soddisfano una condizione.
`tap`	Operatore	Utility	Observable	Esegue effetti collaterali senza modificare il flusso.
`take`	Operatore	Limiting	Observable	Emette solo i primi N valori.
`takeUntil`	Operatore	Limiting	Observable, Notifier	Emette fino a quando un altro Observable emette.
`switchMap`	Operatore	Flattening (Async)	Observable	Sostituisce l'Observable precedente con uno nuovo.
`mergeMap`	Operatore	Flattening (Async)	Observable	Emette i valori di Observables interni in parallelo.
`concatMap`	Operatore	Flattening (Async)	Observable	Emette i valori degli Observables interni in sequenza.
`catchError`	Operatore	Error Handling	Observable	Gestisce errori restituendo un Observable alternativo.
`combineLatest`	Operatore	Combination	Più Observable	Combina i valori più recenti da più Observable.
`distinctUntilChanged`	Operatore	Filtering	Observable	Emette solo se il valore è diverso dal precedente.

8.1.1 Subject, BehaviorSubject e ReplaySubject

In RxJS, `Subject`, `BehaviorSubject` e `ReplaySubject` sono speciali tipi di `Observable` che, oltre a emettere valori, hanno la capacità di gestire e condividere lo stato tra più subscriber. Sono utilizzati per multicast (condividere lo stesso stream di dati tra più abbonati), che differenzia questi tipi di `Subject` dai normali `Observable`.

Un `Subject` è un `Observable` che può emettere valori ai suoi subscriber in modo simile a un normale Observable, ma con una differenza fondamentale: può multicast, cioè inviare lo stesso valore a più subscriber. `Subject` non memorizza valori passati, i nuovi subscriber non ricevono i valori precedenti, ma solo quelli emessi dopo essersi iscritti.

Un `BehaviorSubject` è un tipo di `Subject` che mantiene l'ultimo valore emesso e lo invia a tutti i nuovi subscriber immediatamente quando si iscrivono. `BehaviorSubject` richiede un valore iniziale al momento della creazione. È utile quando si vuole che i subscriber ricevano sempre l'ultimo stato disponibile.

Un `ReplaySubject` è simile a un `BehaviorSubject`, ma anziché memorizzare solo l'ultimo valore emesso, memorizza una cronologia di valori. Quando un nuovo subscriber si iscrive, riceve un numero specificato di valori passati. Questo lo rende utile quando si vuole che i subscriber ricevano una serie di valori precedenti.

Riassumendo:

Tipo	Memorizza l'ultimo valore	Nuovi subscriber ricevono...	Utilizzo tipico
`Subject`	No	Solo valori emessi dopo l'iscrizione	Multicasting senza stato persistente
`BehaviorSubject`	Sì	L'ultimo valore emesso	Gestione dello stato corrente
`ReplaySubject`	Sì (numero definito di valori)	I valori recenti fino al limite specificato	Riprodurre la cronologia degli eventi

8.2 Pipe( )

Il metodo `pipe()` consente di comporre una catena di operatori sui oggetti `Observable`. Ogni operatore all'interno della pipe riceve i dati emessi dall'Observable precedente, li elabora, e li passa all'operatore successivo.

Example

```
import { of } from 'rxjs';
import { map, filter } from 'rxjs/operators';

const obs$ = of(1, 2, 3, 4, 5).pipe(
    filter(x => x % 2 === 0),
    map(x => x * 10)
);

obs$.subscribe(console.log); // Output: 20, 40
```

Nell'esempio precedente, dopo aver creato l'Observable, il metodo `filter()` all'interno di `pipe()` scarta i numeri dispari. Successivamente, il metodo `map()` moltiplica per 10 i numeri rimanenti.

9. Direttive personalizzate

In Angular, le direttive personalizzate sono classi che permettono di modificare il comportamento o l'aspetto degli elementi del DOM a cui sono associate. Si creano attraverso il decoratore `@Directive`.

Example

```
import { Directive, ElementRef, Renderer2, HostListener } from '@angular/core';

@Directive({
selector: '[appEvidenzia]' // uso: <p appEvidenzia>Testo</p>
})
export class EvidenziaDirective {
    constructor(private el: ElementRef, private renderer: Renderer2) {}

    @HostListener('mouseenter') onMouseEnter() {
        this.renderer.setStyle(this.el.nativeElement, 'backgroundColor', 'yellow');
    }

    @HostListener('mouseleave') onMouseLeave() {
        this.renderer.removeStyle(this.el.nativeElement, 'backgroundColor');
    }
}
```

Il codice precedentemente mostrato crea una direttiva personalizzata in Angular per modificare l'aspetto di un elemento HTML al passaggio del mouse. In particolare, questa direttiva imposta uno sfondo giallo quando il cursore si trova sopra l'elemento, e lo rimuove quando il cursore esce. `HostListener` consente di catturare eventi DOM sull'elemento host (come mouseenter, click, ecc.). La variabile `el` rappresenta l'elemento DOM su cui è applicata la direttiva. Il servizio `renderer` consente di modificare il DOM in modo compatibile con le varie piattaforme (es. server-side rendering), infatti permette di impostare lo stile.

Per usare la direttiva in un componente, basta aggiungerla come attributo:

Example

```
<p appEvidenzia>Passa il mouse su questo paragrafo</p>
```

10. Template-driven form

Il Template-driven form in Angular è un approccio per costruire e gestire form in cui la maggior parte della logica è scritta direttamente nell'HTML, sfruttando le direttive offerte da Angular (validator degli input). I form utilizzano il two-way data binding (binding bidirezionale) per aggiornare il modello di dati nel componente quando vengono apportate modifiche nel template, e viceversa.

> **Info**
>
> Angular supporta due approcci di progettazione per i form interattivi:
>
> - I form template-driven ti permettono di usare direttive specifiche per i form direttamente nel template Angular (tratteremo solo questi in questa guida).
> - I form reattivi (reactive forms) forniscono un approccio basato sul modello (model-driven) per la costruzione dei form.
>
> I form template-driven sono un'ottima scelta per form semplici o di piccole dimensioni, mentre i form reattivi sono più scalabili e adatti a form complessi.

10.1 Costruzione

Per costruire un form con approcio Template-drive occore utilizzare le direttive come `ngModel`, `NgModelGroup`, `ngForm` del modulo `FormsModule`. Chiaramente tutto il codice del form sara scritto nel template HTML.

Direttiva	Dettagli
NgModel	Riconcilia le modifiche di valore nell'elemento del form associato con i cambiamenti nel modello dati, permettendo di gestire l'input utente con validazione ed errori.
NgForm	Crea un'istanza di `FormGroup` di livello superiore e la associa a un elemento `<form>` per tracciare i valori aggregati del form e lo stato di validazione. Importando `FormsModule`, questa direttiva è attiva di default su tutti i tag `<form>` non serve aggiungere un selettore speciale.
NgModelGroup	Crea e associa un'istanza di `FormGroup` a un elemento del DOM.

Dopo aver importato il modulo `FormsModule` nel root module o comunque nel modulo che contiene il component in cui si vuole implementare il form, il form può essere implemntato nel seguente modo:

Example

```
<form #f="ngForm" (ngSubmit)="onSubmit(f)">
    <label>
        Nome:
        <input name="nome" [(ngModel)]="utente.nome" required #nome="ngModel" />
    </label>
    <div *ngIf="nome.invalid && nome.touched">
        Il nome è obbligatorio.
    </div>
    <label>
        email:
        <input name="email" [(ngModel)]="utente.email" email />
    </label>

    <h3>Indirizzo</h3>
    <div ngModelGroup="indirizzo">
        <label>
            Via:
            <input name="via" [(ngModel)]="utente.indirizzo.via" required />
        </label>
        <label>
            Città:
            <input name="citta" [(ngModel)]="utente.indirizzo.citta" required />
        </label>
        <label>
            CAP:
            <input name="cap" [(ngModel)]="utente.indirizzo.cap" required />
        </label>
    </div>
    <button type="submit" [disabled]="f.invalid">Invia</button>
</form>
```

Nel ts la funzione `onSubmit()` è cosi definita:

```
export class AppComponent {
    utente = {
        nome: '',
        indirizzo: {
        via: '',
        citta: '',
        cap: ''
        }
    };

    onSubmit(form: any) {
        console.log('Dati inviati:', form.value);
    }
}
```

Dove:

- `#f="ngForm"` : assegna una reference locale al form.
- `[(ngModel)]="utente.nome"` : crea un two-way binding tra l'input e il modello utente.
- `required`, `email`, ecc.: si usano gli attributi HTML per le validazioni.
- `f.invalid` : Angular gestisce automaticamente la validazione e fornisce proprietà come `valid`, `invalid`, `touched`, `pristine`, ecc.
- `ngModelGroup="indirizzo"` crea un gruppo logico per i campi dell'indirizzo. I nomi degli input (es. "via", "citta", ecc.) diventano sotto-proprietà del gruppo indirizzo nel modello del form.

L'oggetto `form.value` avrà questa struttura:

```
{
  nome: 'Mario',
  indirizzo: {
    via: 'Via Roma',
    citta: 'Milano',
    cap: '20100'
  }
}
```