



Introduzione ad Angular 16

A cura di Christian Girardi

Argomenti

- ▶ Variabili Template
- ▶ Creare delle direttive
- ▶ Pipe
- ▶ Service
- ▶ Routing
- ▶ Routing Parametri
- ▶ Redirect ed errori
- ▶ Routing Guard
- ▶ Observable
- ▶ Template Driven
- ▶ Reactive Form
- ▶ modulo HTTP
- ▶ Lazy Loading
- ▶ Build



Variabili Template

Creiamo un input:

```
<input value="test">
```

Vogliamo recuperare il nostro valore da utilizzare nella nostra logica, ma senza usare binding.

Dobbiamo aggiungere quella che viene chiamata variabile di template:

```
<input #inputund value="test">
```

Vediamo come leggere questo valore, nel nostro codice andiamo nel ts.



Variabili Template

Dobbiamo utilizzare un nuovo decoratore:

```
title = 'progettonuovo1';  
@ViewChild('inputuno') valorechevogliousare :any
```

Stesso nome nel html

Abbiamo usato any per prendere il più ampio spettro di possibilità, ma la forma corretta sarebbe:

```
title = 'progettonuovo1';  
@ViewChild('inputuno') valorechevogliousare ! : ElementRef
```

Assicuriamo a ts che
questo elemento non
sarà mai null



Variabili Template

Andiamo a verificare nel onInit ci darà undefined in quanto la view non è ancora inizializzata, per tanto dobbiamo provare a stampare il contenuto in AfterViewInit:


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

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

  title = 'progettonuovo1';
  @ViewChild('inputuno') valorechevogliousare ! : ElementRef

  ngOnInit(): void {}

  ngAfterViewInit(): void {
    console.log(this.valorechevogliousare);
  }
}
```



```
▼ Object { nativeElement: input } app.component.ts
  ► nativeElement: <input _ngcontent-dno-c1="" value="test">
  ► <prototype>: Object { _ }
    ► constructor: class ElementRef {
      constructor(nativeElement) { }
    }
    ► <prototype>: Object { _ }
```

Variabili Template

Inseriamo un bottone per velocizzare le operazioni:

```
<input #inputuno value="test">
<button [(click)]="cliccato()">cliccami</button>
```

```
cliccato(){
  console.log(this.valorechevogliousare);
}
```

```
► Object { nativeElement: input } app
► Object { nativeElement: input } app
► Object { nativeElement: input } app
► Object { nativeElement: input } app
► Object { nativeElement: input } app
```

Troveremo una serie di informazioni , tra tutte quella per noi di interesse è **value**






Variabili Template

Ora per recuperare il contenuto ci basterà :

```
clickato(){  
  console.log(this.valorechevogliousare.nativeElement.value);  
}
```

ALTERNATIVA: Possiamo essere molto più dettagliati aggiungendo:



```
title = 'progettonuovo1';  
@ViewChild('inputuno') valorechevogliousare ! : ElementRef<HTMLInputElement>
```

Ci aiuta in fase di ricerca:

```
clickato(){  
  console.log(this.valorechevogliousare.nativeElement.val);  
}
```

- validationMess... (property) HTMLInputElement.validatio...
- validity
- value
- valueAsData

Direttive

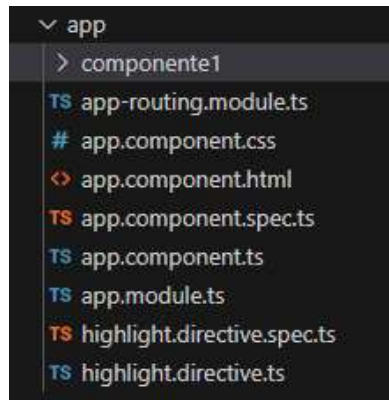
Creeremo una direttiva personalizzata, iniziamo con il digitare nella Command Line:

ng g d highlight

d per directive

```
C:\Users\Christian\Desktop\da cancellare html\angular\progettonuovo1>ng g d highlight
CREATE src/app/highlight.directive.spec.ts (236 bytes)
CREATE src/app/highlight.directive.ts (147 bytes)
UPDATE src/app/app.module.ts (579 bytes)

C:\Users\Christian\Desktop\da cancellare html\angular\progettonuovo1>
```



Direttive

```
angular > progettonuovo1 > src > app > TS highlight.directive.ts > ...  
1  import { Directive } from '@angular/core';  
2    
3  @Directive({  
4    selector: '[appHighlight]'  
5  })  
6  export class HighlightDirective {  
7  
8    constructor() { }  
9  
10 }  
11
```

Ora dobbiamo associare questa direttiva ad un elemento, per tanto modifichiamo il codice come segue:



Direttive

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

@Directive({
  selector: '[appHighlight]'
})
export class HighlightDirective {

  constructor(private element : ElementRef) {
    this.element.nativeElement.style.backgroundColor = 'yellow';
  }
}
```

Proviamo ad applicare questa direttiva ad un elemento

```
<p appHighlight>sono un paragrafo</p>
```

sono un paragrafo



Direttive

Ora dobbiamo gestire il passaggio del mouse, per far questo dobbiamo introdurre un nuovo decoratore HostListener:

```
//  
export class HighlightDirective {  
  constructor(private element : ElementRef) {  
    this.element.nativeElement.style.backgroundColor = 'yellow';  
  }  
  @HostListener()  
}
```



```
  constructor(private element : ElementRef) {  
    this.element.nativeElement.style.backgroundColor = 'yellow';  
  }  
  
  @HostListener('mouseenter') onMouseEnter(){}  
  @HostListener('mouseleave') onMouseLeave(){}  
}
```



Direttive

Ora potremmo fare una cosa del genere:

```
constructor(private element : ElementRef) {}  
  
@HostListener('mouseenter') onMouseEnter(){  
    this.element.nativeElement.style.backgroundColor = 'yellow';  
}  
@HostListener('mouseleave') onMouseLeave(){  
    this.element.nativeElement.style.backgroundColor = 'transparent';  
}  
}
```

Funziona, verificate html , ma possiamo fare di meglio creiamo una funzione cambiacolore():



Direttive

Ecco il codice modificato:

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

@Directive({
  selector: '[appHighlight]'
})
export class HighlightDirective {
  constructor(private element : ElementRef) { }

  @HostListener('mouseenter') onMouseEnter(){
    this.cambiacolore('yellow');
  }

  @HostListener('mouseleave') onMouseLeave(){
    this.cambiacolore('transparent');
  }

  cambiacolore(colore:string){
    this.element.nativeElement.style.backgroundColor = colore;
  }
}
```

sono un paragrafo

Funzionante, ora vediamo di evolvere il codice, otteniamo dati dall'esterno:





Direttive

Per ottenere dati da utilizzare all'interno abbiamo necessita di usare @Input vediamo come:

```
constructor(private element : ElementRef) { }  
  
@Input() appHighlight = '';
```

ATTENZIONE DEVO USARE LO STESSO
NOME DELLA DIRETTIVA
(Osservate il selettore)

Continuiamo la modifica:

```
@Directive({  
  selector: '[appHighlight]'  
})  
export class HighlightDirective {  
  constructor(private element : ElementRef) { }  
  
  @Input() appHighlight = '';  
  
  @HostListener('mouseenter') onMouseEnter(){  
    this.cambiacolore[this.appHighlight];  
  }  
  @HostListener('mouseleave') onMouseLeave(){  
    this.cambiacolore('transparent');  
  }  
  cambiacolore(colore:string){  
    this.element.nativeElement.style.backgroundColor = colore;  
  }  
}
```

Direttive

Sistemiamo HTML:

```
angular > progettonuovo1 > src > app > <img alt="Angular icon" data-bbox="375 325 395 345"/> app.component.html > <img alt="HTML icon" data-bbox="375 365 395 385"/> p  
Go to component  
1 <p [appHighlight]="colore">sono un paragrafo</p>
```

Sistemiamo ts:

```
7  
8 export class AppComponent implements OnInit {  
9  
10 title = 'progettonuovo1';  
11 colore = "green";  
12 }  
13 }
```

Risultato:

sono un paragrafo

**UMANA
FORMA**



Direttive - Esempio

Potremmo per esempio inserire dei radio:

```
<div>
  <input type="radio" name="coloreEvidenziatore" (click)="cambiaColoreEvidenziatore('red')">Rosso
  <input type="radio" name="coloreEvidenziatore" (click)="cambiaColoreEvidenziatore('pink')">Rosa
  <input type="radio" name="coloreEvidenziatore" (click)="cambiaColoreEvidenziatore('purple')">Viola
</div>
```

Ed utilizzare una funzione per cambiare colore:

```
cambiaColoreEvidenziatore(colore: string){
  this.colore = colore
}
```

Otterremo un selettore di colore per la nostra evidenziazione.

Fonte: <https://angular.io/guide/attribute-directives>



Pipe

Le pipe sono funzioni utilizzabili nelle espressioni Stringa.

Slide Esterne



Service

In Angular, i "service" sono una delle caratteristiche fondamentali per organizzare e condividere la logica e i dati tra diversi componenti. Sono classi TypeScript che offrono funzionalità specifiche e possono essere iniettati all'interno di componenti, altri servizi o direttive.

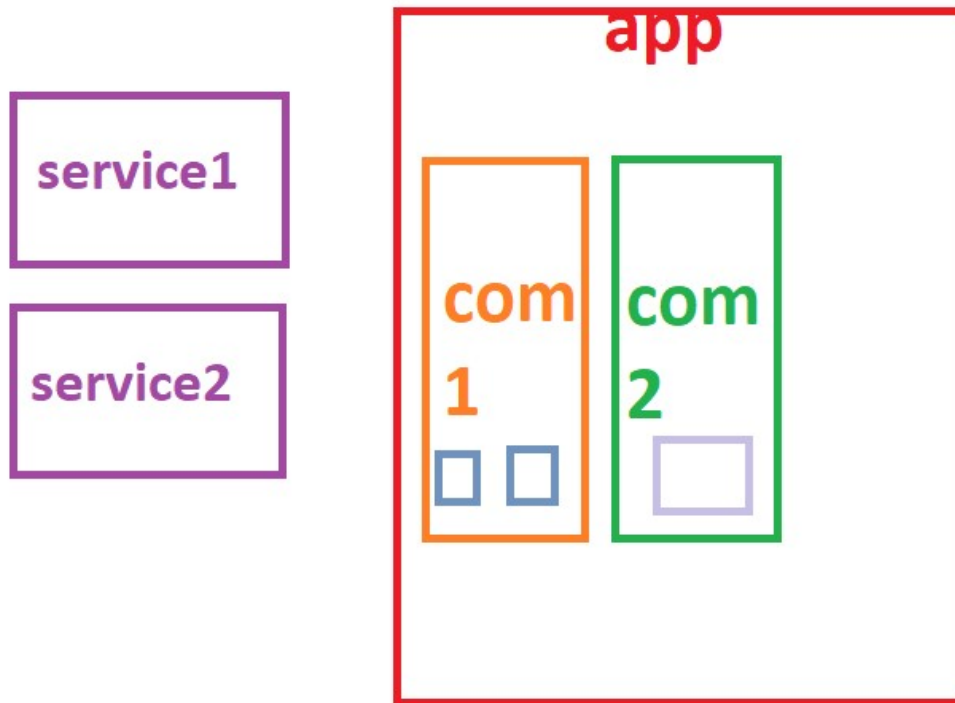
► Ci sono diversi tipi di service in Angular:

1. **Servizi di dati:** Gestiscono la logica di business e l'accesso ai dati. Possono interagire con un server backend, archiviare dati temporanei o condivisi, o eseguire elaborazioni.
2. **Servizi di logica:** Contengono funzionalità che non sono strettamente legate ai dati, come ad esempio funzioni di utilità o metodi per la gestione di flussi di lavoro complessi.
3. **Servizi di autenticazione:** Gestiscono l'autenticazione e l'autorizzazione dell'utente.
4. **Servizi di comunicazione:** Facilitano la comunicazione tra componenti o altre parti dell'applicazione.
5. **Servizi di condivisione di stato:** Mantengono lo stato dell'applicazione condiviso tra più componenti.



Service

Ciò significa che a livelli avanzati, in Angular la logica è conservata solo nei service ed i componenti mandano a video solo oggetti grafici.

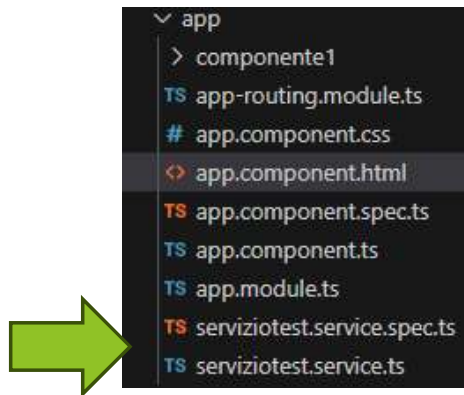


Service

Come prima cosa andiamo a lanciare un nuovo comando ma con struttura simile a quella già vista in passato:

```
ng g s serviziotest
```

Verranno generati alcuni file come sempre:



Apriamo il file service.ts

**UMANA
FORMA**

S ovviamente per service



Service

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

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

  constructor() { }
}
```

```
6 import { Component1Component } from
7
8 @NgModule({
9   declarations: [
10     AppComponent,
11     Component1Component
12   ],
13   imports: [
14     BrowserModule,
15     AppRoutingModule
16   ],
17   providers: [],
18   bootstrap: [AppComponent]
```

@Injectable abbiamo un decoratore che ci informa che stiamo creando un qualcosa di iniettabile nei nostri componenti.

ProvidedIn: ci informa dove è stato reso disponibile, nel caso di root sta a significare ovunque, ma avremmo potuto selezionare i componenti specifici (modo selettivo)

Osserviamo anche **app.module.ts**

Abbiamo anche qui dentro un riferimento al service

Quindi andiamo a scrivere:

```
AppRoutingModule
],
providers: [ServizioTestService],
bootstrap: [AppComponent]
```

N.B. In realtà il modulo è già diffuso in quanto "providedIn: root" quindi dichiararlo qui è superfluo



Service

Nel nostro servizio andremo ad ospitare i dati per la nostra applicazione (**Servizi di dati**), per tanto andiamo a modificare il codice di service.ts:

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

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

  macchine = [
    {modello: 'Porsche 718 Cayman',potenza:'500 CV'},
    {modello: 'Ferrari Daytona SP3',potenza:'840 CV'},
    {modello: 'Audi RS3',potenza:'399 CV'}
  ]

  constructor() { }
```

Il nostro service conterrà
i dati delle macchine

Ora andiamo nel nostro componente1 e proviamo a vedere come accedere a questi dati:



Service

```
ular > progettoonuevo2 > src > app > componente1 > TS componente1
import { Component } from '@angular/core';

@Component({
  selector: 'app-componente1',
  templateUrl: './componente1.component.html',
  styleUrls: ['./componente1.component.css']
})
export class Componente1Component {

}
```

Siamo in ts componente1, per accedere al servizio questo andrà iniettato, dove?

Nel costruttore vediamo come:

```
import { Component } from '@angular/core';
import { ServiziotestService } from '../serviziotest.service';

@Component({
  selector: 'app-componente1',
  templateUrl: './componente1.component.html',
  styleUrls: ['./componente1.component.css']
})
export class Componente1Component {
  constructor(serviziotest : ServiziotestService){
  }
}
```

Aggiungiamo private

Se vi ricordate quando abbiamo visto ts, vi avevo dimostrato la capacità di typescript di farci una scorciatoia nella parte dichiarativa del costruttore, liberandoci al dover utilizzare la classica struttura this.oggetto=oggetto, etc.



Service

Proviamo a vedere se i dati sono disponibili utilizzando onInit:

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

@Component({
  selector: 'app-componente1',
  templateUrl: './componente1.component.html',
  styleUrls: ['./componente1.component.css']
})
export class Componente1Component implements OnInit {
  constructor(private serviziotest : ServiziotestService){

  }
  ngOnInit(): void {
    console.log(this.serviziotest.macchine)
  }
}
```

[componente1.component.ts:14](#)

```
▼ (3) [{...}, {...}, {...}] ⓘ
  ▶ 0: {modello: 'Porsche 718 Cayman', potenza: '500 CV'}
  ▶ 1: {modello: 'Ferrari Daytona SP3', potenza: '840 CV'}
  ▶ 2: {modello: 'Audi RS3', potenza: '399 CV'}
    length: 3
  ▶ [[Prototype]]: Array(0)
```

Angular is running in development mode. Call [enableProdMode\(\)](#) to enable production mode. [core.mjs:23480](#)



Service

Ora per testare la funzionalità dei service provate a fare la stessa cosa su App.
Component:

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent implements OnInit {
  title = 'progettoonuevo2';

  constructor(private serviziotest : ServiziotestService){

  }
  ngOnInit(): void {
    console.log(this.serviziotest.macchine)
  }
}
```

► (3) [{...}, {...}, {...}]

[app.component.ts:16](#)


► (3) [{...}, {...}, {...}]

[componente1.component.ts:14](#)



Service

Possiamo come sempre anche creare dei metodi quindi in service.ts



```
macchine = [  
  {modello:'Porsche 718 Cayman',potenza:'500 CV'},  
  {modello:'Ferrari Daytona SP3',potenza:'840 CV'},  
  {modello:'Audi RS3',potenza:'399 CV'}  
]  
  
constructor() { }  
  
getMacchine(){  
  return this.machines;  
}
```

```
► (3) [{...}, {...}, {...}] app.component.ts:16  
► (3) [{...}, {...}, {...}] component1.component.ts:14
```



Routing

Il routing in Angular è un meccanismo che consente di gestire la navigazione all'interno di un'applicazione a singola pagina (SPA).

In pratica, permette di definire regole per determinare quale componente o vista deve essere visualizzata in base all'URL o ad azioni dell'utente.

Il sistema di routing in Angular si basa su un modulo chiamato RouterModule che fornisce le funzionalità necessarie per mappare gli URL ai componenti corrispondenti.

Ecco alcuni concetti chiave relativi al routing in Angular:



Routing

- ▶ **Routes (Rotte):** Le rotte definiscono la mappatura tra un URL specifico e un componente da caricare quando quell'URL è raggiunto. Ogni rotta ha un percorso (path) e, facoltativamente, può avere parametri o wildcard per gestire URL dinamici.
- ▶ **Router Outlet:** È una direttiva Angular (<router-outlet>) utilizzata nei template dei componenti per indicare il luogo in cui i componenti associati alle rotte verranno visualizzati all'interno dell'applicazione.
- ▶ **RouterModule:** È un modulo Angular che fornisce i servizi di routing e il supporto per definire e gestire le rotte dell'applicazione.
- ▶ **Route Parameters (Parametri di rotta):** Consentono di passare dati specifici attraverso l'URL, ad esempio identificatori di risorse o altri parametri necessari per visualizzare correttamente una determinata vista.
- ▶ **RouterLink:** È una direttiva Angular utilizzata per creare link tra le diverse viste dell'applicazione, specificando le rotte associate agli URL.



Routing

Un esempio di configurazione delle rotte:

```
import { NgModule } from '@angular/core';
import { RouterModule, Routes } from '@angular/router';
import { HomeComponent } from './home.component';
import { AboutComponent } from './about.component';

const routes: Routes = [
  { path: '', redirectTo: '/home', pathMatch: 'full' },
  { path: 'home', component: HomeComponent },
  { path: 'about', component: AboutComponent }
];

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



Routing

Nel caso il vostro file app-routing non fosse presente esiste un'istruzione per farlo generare automaticamente:

```
ng g module app-routing --flat --module=app
```

Aggiungete `<router-outlet></router-outlet>` in `app.component.html`

Andiamo a vedere il file `app-routing.module.ts`, per far funzionare le rotte dobbiamo andare a agire nella parte dedicata appunto alle rotte:

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

const routes: Routes = [];

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



Routing

Andiamo a vedere la struttura base per definire una rotta:

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

N.B. Ricordatevi di aggiungere
`<router-outlet></router-outlet>`
In app.component.html

Serve per far
visualizzare i
componenti delle
rotte!!!!

Per una questione pratica andiamo a generare un paio di componenti :

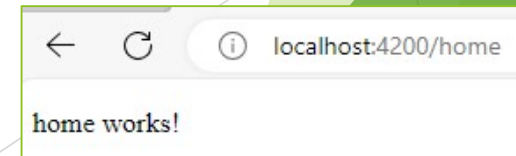
- ▶ Home
- ▶ Contatti
- ▶ Servizi

```
> contatti  
> home  
> servizi  
TS app-routing.module.ts
```

Fatto questo predisponiamo le rotte

```
const routes: Routes = [  
  {path: 'home', component: HomeComponent },  
  {path: 'servizi', component: ServiziComponent },  
  {path: 'contatti', component: ContattiComponent }  
];
```


**UMANA
FORMA**



Routing

Se vogliamo essere precisi possiamo modificare:

```
const routes: Routes = [  
  {path: 'home', component: HomeComponent },  
  {path: 'servizi', component: ServiziComponent },  
  {path: 'contatti', component: ContattiComponent }  
];
```



```
const routes: Routes = [  
  {path: '', component: HomeComponent },  
  {path: 'servizi', component: ServiziComponent },  
  {path: 'contatti', component: ContattiComponent }  
];
```

Così da settare la home all'apertura della prima pagina del nostro server.

Possiamo inserire degli elementi `<a>` nelle nostre pagine ma per evitare il ricaricamento della pagina E' OBBLIGATORIO utilizzare una forma di collegamento che ci viene messa a disposizione da Angular, ovvero:

```
<a [routerLink]="['/percorso']"> testo link </a>
```





Routing Parametri

Andiamo a lavorare su un componente specifico: contatti inoltre utilizzeremo i service precedentemente visti per avere dei dati.

Andiamo su contatti ed iniettiamo il service per i dati:

```
import { Component } from '@angular/core';
import { ServiziotestService } from '../serviziotest.service';

@Component({
  selector: 'app-contatti',
  templateUrl: './contatti.component.html',
  styleUrls: ['./contatti.component.css']
})
export class ContattiComponent {
  constructor(private serviziotest : ServiziotestService){

  }
  ngOnInit(): void {
    console.log(this.serviziotest.getMacchine())
  }
}
```

Routing Parametri

Ora vogliamo utilizzare questi dati (ngfor) quindi li scarichiamo in una variabile che poi utilizzeremo:

```
})  
export class ContattiComponent {  
  
  macchine:any;  
  
  constructor(private serviziotest : ServiziotestService){  
  
  }  
  ngOnInit(): void {  
    console.log(this.serviziotest.getMacchine())  
    this.macchine= this.serviziotest.getMacchine();  
  }  
}
```



Routing Parametri

Ora mandiamo a video su contatti.component.html:

```
Go to component  
<p>contatti works!</p>  
<div *ngFor="let macchina of macchine">  
  <p>{{macchina.modello}}</p>  
</div>
```



```
contatti works!  
Porsche 718 Cayman  
Ferrari Daytona SP3  
Audi RS3
```

Ora per aggiungere il parametro, torniamo su app-routing.module e andiamo ad aggiungere:



```
const routes: Routes = [  
  {path: '', component: HomeComponent },  
  {path: 'servizi', component: ServiziComponent },  
  {path: 'contatti', component: ContattiComponent },  
  {path: 'contatti/:id', component: ContattiComponent },  
];
```



Routing Parametri

Ora dobbiamo recuperare il parametro in caso venga richiesto nell'URL, quindi torniamo in contatti.module.ts:



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

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

  macchine:any;

  constructor(private serviziotest : ServiziotestService, private route: ActivatedRoute){
```

Abbiamo bisogno di activated route per gestire la rotta attiva, in questo momento:



Routing Parametri

Recuperiamo il parametro:

```
}  
ngOnInit(): void {  
  console.log(this.serviziotest.getMACchine())  
  this.macchine= this.serviziotest.getMACchine();  
  console.log( this.route.snapshot.paramMap.get('id') );  
}
```

Utilizzo paramMap , perché
potrei avere piu parametri ad
es: `:/id/:nome/:cognome`

2

[contatti.component.ts:20](#)



Redirect ed errori

Andremo a gestire il famigerato 404 page not found, vediamo come, iniziamo creando un componente per gestire l'errore:

ng g c errore

Aggiungiamo il path nell'app-router

```
const routes: Routes = [  
  {path: '', component: HomeComponent },  
  {path: 'servizi', component: ServiziComponent },  
  {path: 'contatti', component: ContattiComponent },  
  {path: 'contatti/:id', component: ContattiComponent },  
  {path: '404', component: ErroreComponent}]
```


Ora sempre qui aggiungiamo:

```
{path: '', component: HomeComponent },  
{path: 'servizi', component: ServiziComponent },  
{path: 'contatti', component: ContattiComponent },  
{path: 'contatti/:id', component: ContattiComponent },  
{path: '404', component: ErroreComponent},  
{path: '**', redirectTo: '/404'}
```



Focus : Home page

La forma corretta per la gestione della home page è questa:



```
const routes: Routes = [  
  {path: '', pathMatch: 'full', redirectTo: '/home' },  
  {path: 'home', component: HomeComponent },  
  {path: 'servizi', component: ServiziComponent },  
  {path: 'contatti', component: ContattiComponent },  
  {path: 'contatti/:id', component: ContattiComponent },  
  {path: '404', component: ErroreComponent},  
  {path: '**', redirectTo: '/404'}  
];
```



Routing Guard

Con le guardie possiamo proteggere le pagine, vediamo come:

Per prima cosa dobbiamo realizzare un servizio di autorizzazione (auth service)

ng g s auth

```
TS auth.service.spec.ts
TS auth.service.ts
TS serviziotest.service.spec.ts
TS serviziotest.service.ts
```

Inseriamo una variabile che poi modificheremo per dire che l'utente è loggato o no partiamo con un false

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

@Injectable({
  providedIn: 'root'
})
export class AuthService {
  utenteLoggato = false;

  constructor() { }
```



Routing Guard

Creiamo anche un metodo che ci ritornerà lo stato della variabile:

```
utenteLog(){  
  return this.utenteLoggato;  
}
```

Ora introduciamo la guardia effettiva, indovinate come?

ng g guard auth

Che ci chiederà:

```
? Which type of guard would you like to create? (Press <space> to select, <a> to toggle all, <i> to invert selection, and <enter> to proceed)  
>(*) CanActivate  
  ( ) CanActivateChild  
  ( ) CanDeactivate  
  ( ) CanLoad  
  ( ) CanMatch
```

Selezioniamo la prima CanActivate.

**UMANA
FORMA**

```
TS app.module.ts  
TS auth.guard.spec.ts  
TS auth.guard.ts
```



Routing Guard

```
import { Injectable } from '@angular/core';
import { ActivatedRouteSnapshot, CanActivate, RouterStateSnapshot, UrlTree } from '@angular/router';
import { Observable } from 'rxjs';

@Injectable({
  providedIn: 'root'
})
export class AuthGuard implements CanActivate {
  canActivate(
    route: ActivatedRouteSnapshot,
    state: RouterStateSnapshot): Observable<boolean | UrlTree> | Promise<boolean | UrlTree> | boolean | UrlTree {
    return true;
  }
}
```

Importiamo il nostro servizio aut nel costruttore come già visto in precedenza:

```
import { AuthService } from '../auth.service';

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


  canActivate(
```





Routing Guard

La cosa fondamentale per il funzionamento è che la guardia ritorni lo stato dell'autenticazione per tanto , abbiamo collegato il servizio possiamo verificare lo stato dell'utente e ritornarlo modifichiamo quindi:



```
@Injectable({
  providedIn: 'root'
})
export class AuthGuard implements CanActivate {

  constructor(private authService : AuthService){}

  canActivate(
    route: ActivatedRouteSnapshot,
    state: RouterStateSnapshot): Observable<boolean | UrlTree> | Promise<boolean | UrlTree> | boolean | UrlTree {
    return this.authService.utenteLog();
  }
}
```

Torniamo su routing module

Routing Guard

Andiamo su servizi e modifichiamo:

```
{path: home , component: HomeComponent },  
{path: 'servizi', component: ServiziComponent, canActivate:[AuthGuard] },  
{path: 'contatti' , component: ContattiComponent }
```

Array di guardie

CanActivate attiva la rotta solo se la guardia ritornerà true, quindi avviamo il server e proviamo ad andare su servizi. Il link risulterà inattivo.

Proviamo a modificare la variabile in true

Il link ricomincerà a funzionare normalmente



Observable

Gli Observable in Angular sono parte integrante del modulo RxJS (Reactive Extensions for JavaScript) che offre un'implementazione della programmazione reattiva basata sugli Observable.

Un Observable è una sequenza di valori che può essere emessa nel tempo. Può rappresentare eventi asincroni come richieste HTTP, eventi del DOM, dati da un database e molto altro ancora. Gli Observable possono anche trasmettere errori o notificare che l'operazione è completata con successo.

1. **Emittenti di dati:** Gli Observable emettono sequenze di valori. Questi valori possono essere emessi singolarmente o come flussi di dati nel tempo.
2. **Asincronia:** Gli Observable gestiscono operazioni asincrone, consentendo di reagire ai dati non appena sono disponibili, senza bloccare il thread principale dell'applicazione.
3. **Metodi di operazione:** Gli Observable offrono una vasta gamma di operatori che consentono di modificare, filtrare, trasformare e combinare i flussi di dati in modi diversi senza mutare i dati originali.
4. **Gestione degli errori e completamento:** Gli Observable possono gestire errori che si verificano durante l'emissione dei dati e notificare quando un flusso è completo.



Observable

In Angular, gli Observable vengono ampiamente utilizzati per gestire richieste asincrone come chiamate HTTP, eventi del DOM, aggiornamenti di stato e altro ancora. Ad esempio, quando si effettua una chiamata HTTP per recuperare dati da un server, il metodo HttpClient di Angular restituisce un Observable che può essere sottoscritto per ottenere i dati quando sono pronti, e cancellato (unsubscribe) quando non più necessario.

Esempio di utilizzo di un Observable con una chiamata HTTP in Angular:

```
import { HttpClient } from '@angular/common/http';
import { Observable } from 'rxjs';

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

  fetchData(): Observable<any> {
    return this.http.get('https://api.example.com/data');
  }
}
```

Nell'esempio, fetchData() è un metodo del service DataService che esegue una chiamata HTTP GET a un endpoint. Il metodo get() restituisce un Observable che può essere sottoscritto da altri componenti per ottenere i dati quando la chiamata HTTP ha successo.



Observable

Differenza fra OBSERVABLE ed un PROMISE in Angular?

- ▶ Entrambi servono a gestire eventi asincroni come ad esempio una chiamata al server. La Promise attende che una attività parallela precedentemente lanciata termini o fallisca. Quindi una Promise attende l'esito di un solo evento per poi scatenare su di esso altri calcoli.
- ▶ L'osservabile invece è un tubo nel quale possono essere immessi più eventi in continuazione, e ognuno di essi verrà processato allo stesso modo una volta entrato nel tubo e dopo aver atteso il suo turno.





Template Driven

Il template driven è un form che viene gestito completamente dal nostro template.

Inseriamo un form in un nostro componente

Form senza action non dobbiamo inviare!!!

```
<form>  
  <label for="fname">nome:</label><br>  
  <input type="text" id="fname" name="fname" value="Nome"><br>  
  <label for="lname">Cognome:</label><br>  
  <input type="text" id="lname" name="lname" value="Cognome"><br><br>  
  <input type="submit" value="Submit">  
</form>
```

**UMANA
FORMA**

servizi works!

nome:

Cognome:

Template Driven

Aggiungiamo una variabile template per il nostro form, come visto nelle prime slide:

```
<form #formPrincipale>
  <label for="fname">nome:</label><br>
  <input type="text" id="fname" name="f
```


Verifichiamo che in app.module.ts sia presente :

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



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

```
@NgModule({
  declarations: [
    AppComponent,
    Componente1Component,
    HomeComponent,
    ContattiComponent,
    ServiziComponent,
    ErroreComponent
  ],
  imports: [
    BrowserModule,
    AppRoutingModule, FormsModule
  ],
})
```



Template Driven


Torniamo nel nostro form ed ora aggiungiamo:




```
<form #formPrincipale = "ngForm">
  <label for="fname">nome:</label><br>
  <input type="text" id="fname" name="fname" value="Nome"><br>
  <label for="lname">Cognome:</label><br>
  <input type="text" id="lname" name="lname" value="Cognome"><br><br>
  <input type="submit" value="Submit">
</form>
```

Stiamo comunicando ad angular che questo form sarà gestito da lui e non dal html.

A questo punto si aprono un sacco di opzioni fra cui il direzionamento del submit verso qualcosa, creiamo un metodo onSubmit() e poi modifichiamo il form



```
onSubmit(){
}
```



```
<form #formPrincipale = "ngForm" (ngSubmit)="onSubmit()">
  <label for="fname">nome:</label><br>
```



Template Driven

Ora passiamo il form alla nostra funzione, modifichiamo quanto appena fatto:

```
<form #formPrincipale = "ngForm" (ngSubmit)="onSubmit({formPrincipale})">  
  <label for="fname">nome:</label><br>
```

formPrincipale è il nome della mia variabile template che passo alla funzione

Modifichiamo anche la funzione :

```
onSubmit(form :any){  
}
```

Possiamo per esempio inserire un console.log

Torniamo a concentrarci sul nostro form e modifichiamo:



Template Driven

Andiamo a modificare il nostro form, in maniera massiva:

```
<form #formPrincipale="ngForm" (ngSubmit)="onSubmit(formPrincipale)">
  <label for="fname">nome:</label><br>
  <input type="text" id="fname" name="fname" ngModel required><br>
  <label for="femail">email:</label><br>
  <input type="email" id="femail" name="femail" ngModel required email><br>
  <label for="lname">Cognome:</label><br>
  <input type="text" id="lname" name="lname" ngModel required><br><br>
  <button type="submit" [disabled]="!formPrincipale.valid">Invia</button>
</form>
```

Aggiungendo ngModel abbiamo informato angular che deve considerare i campi in oggetto , infatti possiamo specificare ad esempio i campi obbligatori, il tipo

Ed in aggiunta alla fine facciamo un controllo form per abilitare o meno il bottone di submit.



```
▼ _rawValidators: []
  _rawValidators: []
  asyncValidator: (...)
  control: (...)
  controls: (...)
  dirty: (...)
  disabled: (...)
  enabled: (...)
  errors: (...)
  formDirective: (...)
  invalid: (...)
  path: (...)
  pending: (...)
  pristine: (...)
  status: (...)
  statusChanges: (...)
  touched: (...)
  untouched: (...)
  valid: (...)
  validator: (...)
  value: Object
    femail: "q"
    fname: "q"
    lname: "q"
```

Template Driven

Possiamo poi verificare che tipo di classe viene assegnata in caso di controllo non valido:

```
<form _ngcontent-ebw-c2 novalidate class="ng-pristine ng-invalid ng-touched">
  <label _ngcontent-ebw-c2 for="fname">nome:</label>
  <br _ngcontent-ebw-c2>
  <input _ngcontent-ebw-c2 type="text" id="fname"
    required ng-reflect-required ng-reflect-name="fname"
    class="ng-pristine ng-invalid ng-touched">
```

Proviamo a creare una class CSS

```
.ng-invalid{
  background-color: red;
}
```

servizi works!

nome:

email:

Cognome:

```
1 input.ng-invalid{
2   background-color: red !important;
3 }
```

servizi works!

nome:

email:

Cognome:



UMANA
FORMA



UMANA
FORMA



UMANA
FORMA



UMANA
FORMA



UMANA
FORMA



UMANA
FORMA



UMANA
FORMA

