ITALDATA

TYPESCRIPT

www.italdata.it

ANGULAR I

autore Gino Visciano

Codice: ANGV0118

Sommario

- Introduzione
- Binding e Comunicazione
- Ciclo di vita di un'Applicazione Angular
- Dipendence Injection
- Form Template Driven
- Form Reactive

Introduzione

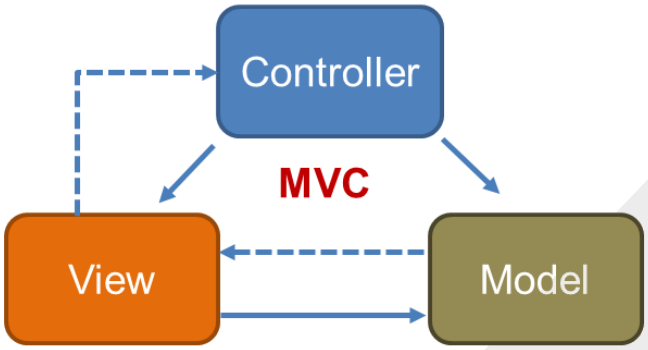
1/43 - Che cos'è Angular

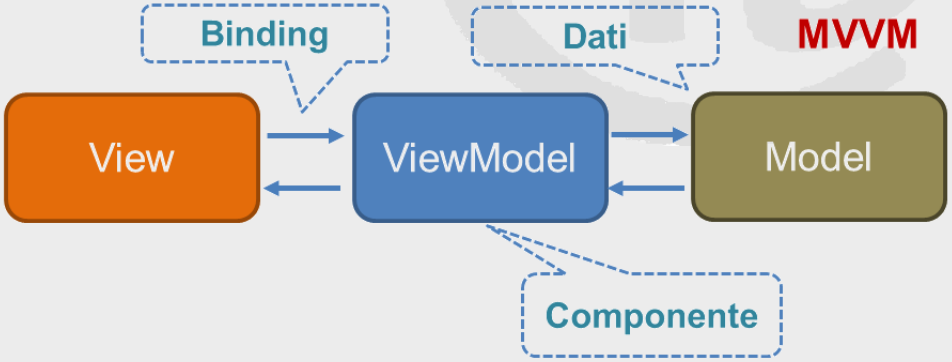
Angular è un **framework** client-side che permette di creare Applicazioni **Sigle Page**, sviluppato da Google nel 2010, è un ambiente di sviluppo open source che si basa su **HTML**, **CSS**, **JavaScript** e **TypeScript**.

Le applicazioni Angular sono di tipo **Front-End** e possono girare unicamente su **client Web**..

La caratteristica più interessante di Angular è il **Data Binding** bidirezionale che permette di collegare le **viste** al **codice** e viceversa.

Il modello implementato da un'applicazione Angular è di tipo **MVVM** (**M**odel **V**iew **V**iew**M**odel).
In questo tipo di modello il controller, classico del modello **MVC** (**M**odel **V**iew **C**ontroller) usato da **Java**, **C#** e **PHP**, viene sostituito dal **ViewModel** (Componente) che attraverso il codice mette in relazione **view** e **model**.



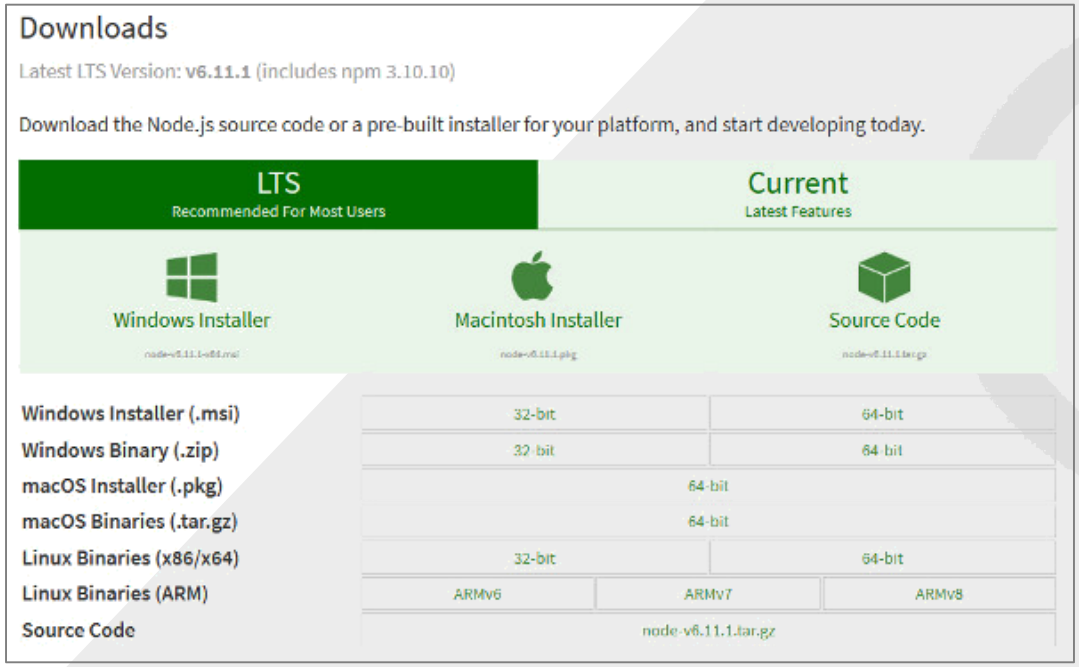


Introduzione

2/43 - L'Ambiente di Sviluppo

Per lavorare con **Angular** bisogna installare **Node.js**, scegliete la versione compatibile con il sistema operativo della macchina dove verrà usato.

Per scaricare **Node.js** dovete collegarvi al sito https://nodejs.org/it/, selezionare la voce di menu **download** e scaricare il file d'installazione corrispondente alla vostra piattaforma di lavoro, come mostra l'immagine seguente:

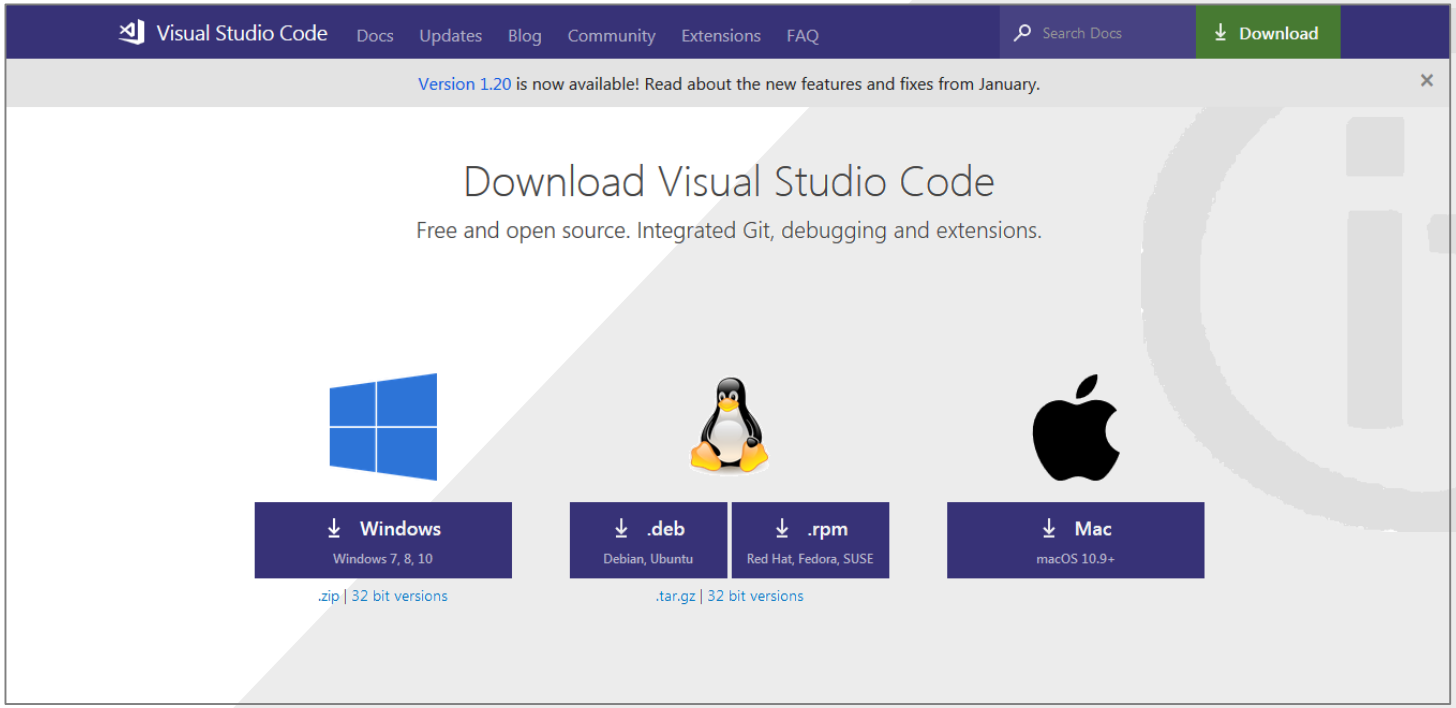


Introduzione

3/43 - L'Ambiente di Sviluppo

Per scrivere le applicazioni potete usare come **IDE** (**I**ntegrated **D**evelopment **E**nvironment) **Visual Studio Code**.

Per scaricare Visual Studio Code dovete collegarvi al sito https://code.visualstudio.com/download, e scaricare il file d'installazione corrispondente alla vostra piattaforma di lavoro, come mostra l'immagine seguente:



autore Gino Visciano

Introduzione

4/43 - Come creare un'Applicazioni Angular

Per creare un'applicazione **Angular** potete usare il comando:

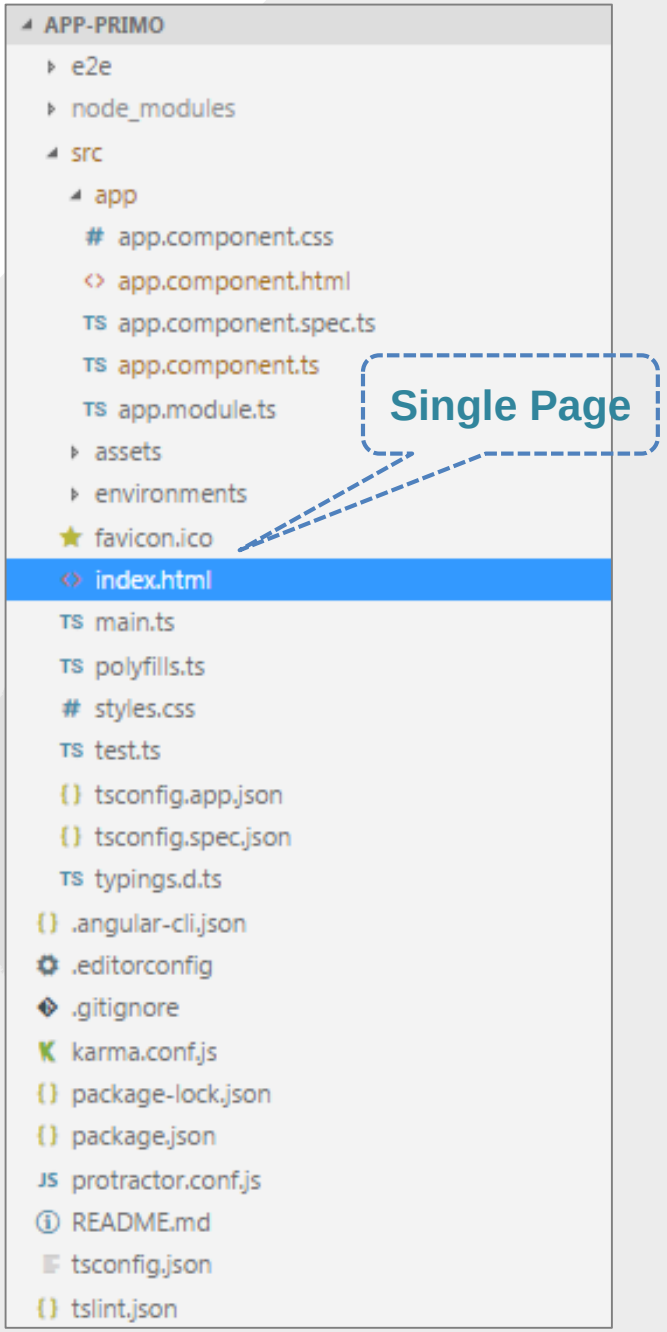
ng new app-prima

Attenzione il nome di un'applicazione Angular non può contenere né spazi né undescore _.
Dopo l'esecuzione del comando la struttura dell'applicazione Angular sarà quella che vedete nell'immagine a destra.

Index.html

```
<!doctype html>
<html lang="en">
<head>
<meta charset="utf-8">
<title>app-mia</title>
<base href="/">
<meta name="viewport" content="width=device-width, initial-scale=1">
<link rel="icon" type="image/x-icon" href="favicon.ico">
</head>
<body>
<app-root></app-root>
</body>
</html>
```

Selettore



Introduzione

5/43 - Come generare gli elementi di un'Applicazioni Angular

Per generare gli elementi ce servono per creare un'applicazione **Angular** potete usare il comando:

ng generate [nome]

[nome]:

- ✓ class
- ✓ component
- ✓ directive
- ✓ enum
- ✓ guard
- ✓ interface
- ✓ module
- ✓ pipe
- ✓ service

Ad esempio per creare un componente:

ng generate component mio-componente

File generati nella cartella src/app:

mio-componente.component.css
mio-componente.component.html
mio-componente.component.spec.ts
mio-componente.component.ts

Introduzione

6/43 - Struttura di un'applicazione Single Page

Angular è un **framework** che permette di creare Applicazioni **Sigle Page**.
Single Page significa che un'applicazione Angular per visualizzare le informazioni utilizza la stessa **pagina HTML** che cambia dinamicamente attraverso l'aggiornamento di **viste HTML** associate a **selettori** inseriti nella pagina.

La **pagina HTML** caricata inizialmente in Angular (**Sigle Page**) è **Index.html**. Il selettore **<app-root></app-root>** permette di caricare la vista inserita nel file **app.component.html**.

L'associazione tra il **selettore <app-root></app-root>** e la **vista** creata nel file **app.component.html** avviene all'interno del **decoratore @component** presente nel componente **app.component.ts**, come mostra l'esempio seguente:

```
import { Component } from '@angular/core';
@Component({
selector: 'app-root',
templateUrl: './app.component.html',
styleUrls: ['./app.component.css']
})
export class AppComponent { … }
```

Decoratore

Selettore

Vista

Logica/codice Vista

Binding e Comunicazione

7/43 - Interpolazione

L' **Interpolazione** è una tecnica di **binding unidirezionale**, permette di **iniettare** dati in una vista. Per implementare questo meccanismo basta inserire nella vista il nome di una variabile oppure di una funzione definita nel **componente** associato alla vista, all'interno di una coppia di parentesi graffe aperte e chiuse {{ … }}. Vediamo un semplice esempio:

```
// app.component.ts
import { Component } from '@angular/core';
@Component({
selector: 'app-root',
templateUrl: './app.component.html',
styleUrls: ['./app.component.css']})
export class AppComponent {
titolo = 'Interpolazione';
msg:string='Somma';
constructor(){}
formula():string{
return '2+2';}
tre():number{
return 3;}}
```

```
<!-- app.component.html -->
<h3>{{titolo}}</h3>
<h4>
{{msg+' '+ formula()}}={{1+tre()}}
</h4>
```

Binding e Comunicazione

8/43 - Interpolazione

Per svolgere l'esercizio indicato nell'esempio precedente nel modo seguente:

1) Create una cartella angular ed attivatela
2) Eseguite il comando: ng new app-somma
3) Modificate il componente app.component.ts, con il codice fornito nell'esempio
4) Modificate la vista app.component.html , con il codice fornito nell'esempio
5) Lanciate il server **node** con il comando: ng serve
6) Avviate un Browser e caricate l'applicazione utilizzando l'**url**: localhost:4200.

Dovreste ottenere il risultato seguente:

Output

```
Primo Esercizio

Somma 2+2=4
```

Binding e Comunicazione

9/43 - Variabili di riferimento

Le **variabili di riferimento** sono nomi che si assegnano ai **tag HTML** per creare il **binding unidirezionale** tra la **vista** ed il **componente** associato, di verso opposto all'**Interpolazione**.
Una variabile di riferimento è semplicemente un nome preceduto dal cancelletto #, come mostra l'esempio seguente:

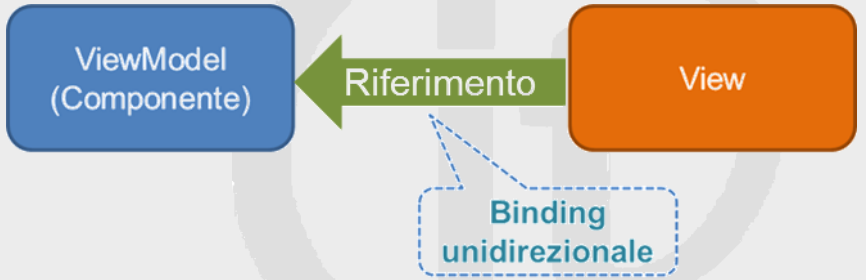
```
<input type='text'  name='cognome' id='cognome' #rif_cognome >
```

In questo caso il valore **rif_cognome.value** conterrà il valore della casella di testo **cognome**.
Per capire meglio come funzionano le variabili di riferimento facciamo un esempio.

```
<!-- app.component.html -->
<h3>
{{titolo+":"+data}}
</h3>
<div>
<input #rif_nome placeholder="il tuo nome"><br>
<input #rif_cognome placeholder="il tuo cognome"><br>
<input #rif_eta placeholder="la tua età"><br>
<button (click)="invioDati(rif_nome.value,rif_cognome.value,rif_eta.value)">invia</button>
<p>Persona: {{nome.+", "+cognome+", "+eta}}</p>
</div>
```

Variabili interpolate

Variabili di riferimento



Variabili interpolate

autore Gino Visciano

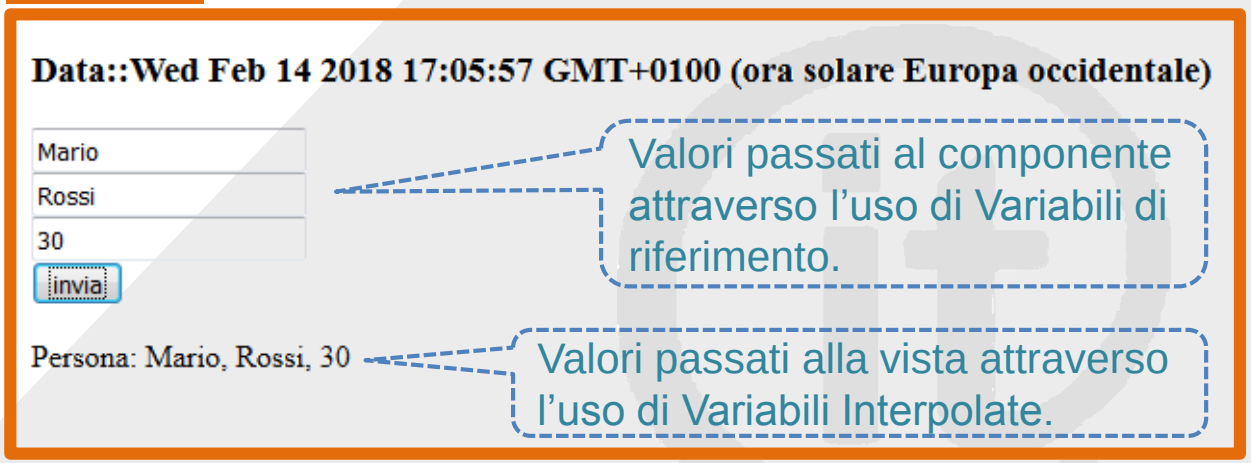
Binding e Comunicazione

10/43 - Variabili di riferimento

I valori delle **variabili di riferimento**, attraverso l'evento **click**, vengono inviati dalla vista alla funzione **invioDati** del componente. La funzione li assegna alle **variabili interpolate nome**, **cognome** ed **eta** con l'obiettivo di visualizzare nella stessa vista i valori passati al componente.

```
import { Component } from '@angular/core';
@Component({
selector: 'app-root',
templateUrl: './app.component.html',
styleUrls: ['./app.component.css']})
export class AppComponent {
titolo:string = 'Data:';
data:Date;
nome:string;
cognome:string;
eta:number;
constructor() {this.data=new Date();}
invioDati(nome:string, cognome:string, eta:number){
this.nome=nome;
this.cognome=cognome
this.eta=eta;}}
```

Output



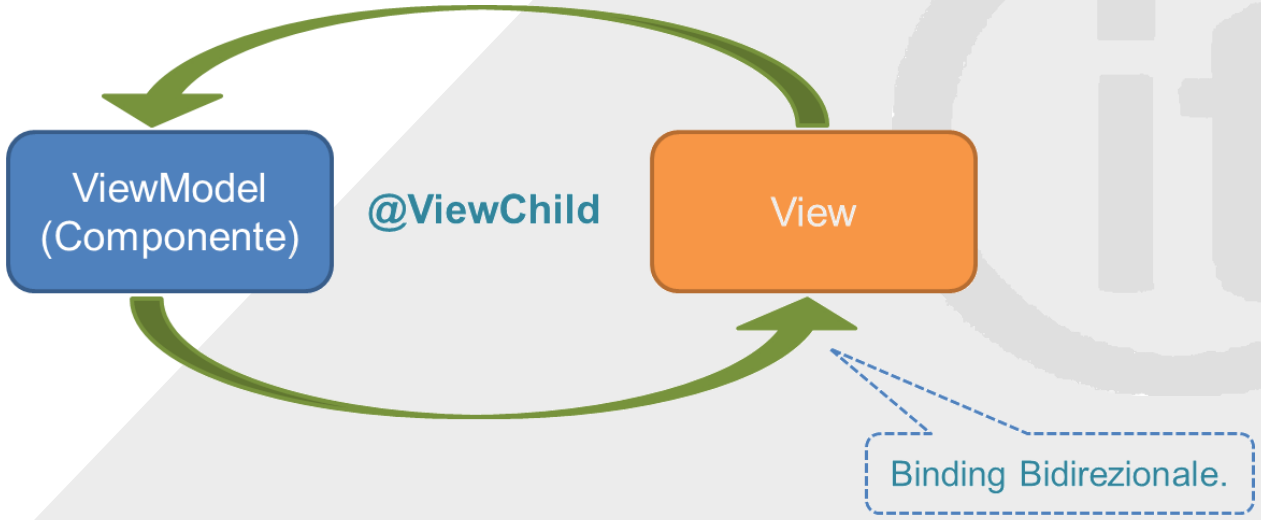
autore Gino Visciano

Binding e Comunicazione

11/43 - @ViewChild

Il decoratore **@ViewChild** si usa per catturare un elemento del **DOM** (**D**ocument **O**bject **M**odel) della vista, per gestirlo nell'applicazione.

In questo caso il **binding** è di tipo **bidirezionale,** perché la comunicazione tra la **vista** ed il **componente** associato avviene in entrambi i versi.



Binding e Comunicazione

12/43 - @ViewChild

Per usare il decoratore **@ViewChild** lo dovete importare nel **componente** associato alla **vista** con cui intendete comunicare, come mostra l'esempio seguente:

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

Oltre al **decoratore** serve anche la classe **ElementRif**, perché i **tag** della **vista**, essendo oggetti del **DOM**, restituiscono riferimenti di questo tipo.

Dopo aver importato il **decoratore** potete eseguire il **binding** con i **tag** della **vista**, utilizzando le **variabili di riferimento** associate, come mostra l'esempio seguente:

```
@ViewChild('rif_nome') private tagNome:ElementRef;
```

Comando da inserire nel Componente

Riferimento al Tag input

```
<input #rif_nome placeholder="nome della persona" (input)="inputValori()">
```

Tag nella Vista

autore Gino Visciano

Binding e Comunicazione

13/43 - @ViewChild

Nell'esempio seguente con il decoratore **@ViewChild** leggiamo i dati della **persona** inseriti nella **scheda** della vista. Nel **componente**, i dati letti, vengono assegnati alle **variabili interpolate** per visualizzarli nella tabella. Cliccando sul pulsante **Pulisci campi** viene pulita la **scheda** e la **tabella**.

```
<!-- app.component.html -->
<p>{{titolo}}</p>
<input #rif_nome placeholder="nome della persona" (input)="inputValori()"><br>
<input #rif_cognome placeholder="cognome della persona" (input)="inputValori()"><br>
<input #rif_eta placeholder="eta della persona" (input)="inputValori()"><br>
<button (click)="pulisciValori()">Pulisci campi</button>
<br><br>
<table border="1">
<tr><th width="150px">Nome</th><th width="150px">Cognome</th><th width="50px">Età</th></tr>
<tr><td>{{nome}}</td><td>{{cognome}}</td><td align="right">{{eta}}</td></tr>
</table>
```

Scheda

(click)=Evento che ad ogni click esegue pulisciValuori().

(input)=Evento che ad ogni cambiamento esegue inputValuori().

Tabella con Variabili Interpolate

autore Gino Visciano

Binding e Comunicazione

14/43 - @ViewChild

```
//app.component.ts
import { Component, ViewChild, ElementRef} from '@angular/core';
@Component({
selector: 'app-root',
templateUrl: './app.component.html',
styleUrls: ['./app.component.css']})
export class AppComponent {
  titolo = 'Inserisci persona:';
  @ViewChild('rif_nome') private tagNome:ElementRef;
  @ViewChild('rif_cognome') private tagCognome:ElementRef;
  @ViewChild('rif_eta') private tagEta:ElementRef;
  private nome:string="";
  private cognome:string="";
  private eta:number=0;
  inputValori(){
    this.nome=this.tagNome.nativeElement.value;
    this.cognome=this.tagCognome.nativeElement.value;
    this.eta=this.tagEta.nativeElement.value;}
  pulisciValori(){
    this.tagNome.nativeElement.value="";
    this.tagCognome.nativeElement.value="";
    this.tagEta.nativeElement.value=0;
    this.inputValori();}}
```

Vista associata al componente.

Binding Bidirezionale

Assegnazione valori variabili Interpolate

Pulizia scheda e tabella

Binding e Comunicazione

15/43 - @ViewChild

Per svolgere l'esercizio indicato nell'esempio precedente nel modo seguente:

1) Create una cartella angular ed attivatela
2) Eseguite il comando: ng new app-persona
3) Modificate il componente app.component.ts, con il codice fornito nell'esempio
4) Modificate la vista app.component.html , con il codice fornito nell'esempio
5) Lanciate il server **node** con il comando: ng serve
6) Avviate un Browser e caricate l'applicazione utilizzando l'**url**: localhost:4200.

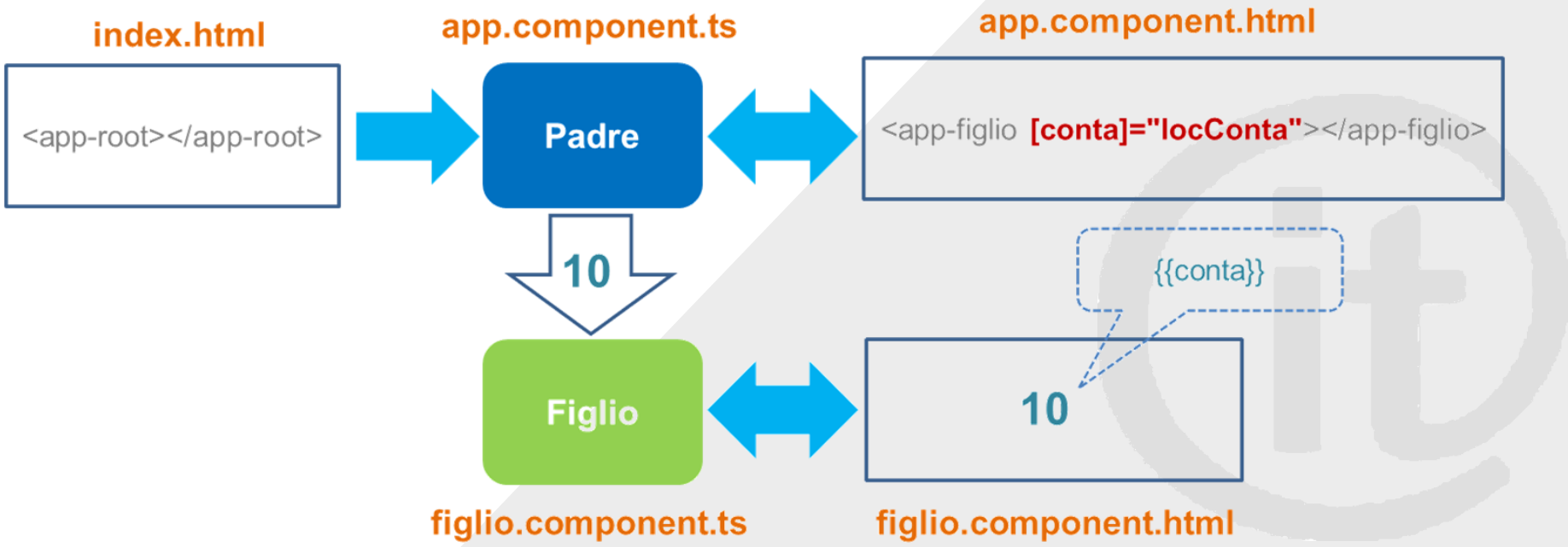
Dovreste ottenere il risultato seguente:

Output

Binding e Comunicazione

16/43 - @Input

Il decoratore **@Input** può essere usato per passare dati da un componente padre ad un componente figlio, come mostra lo schema seguente:



Binding e Comunicazione

17/43 - @Input

L'esempio seguente mostra in che modo usare il decoratore @Input per incrementare dal componente padre un contatore presente nella vista del componente figlio.:

```
<!--app.component.html-->
{{titolo}}<br>
<p>Vista (Padre)<br>
<button (click)="incrementaContatore()">aggiungi 1 al figlio</button></p>
<hr>
<app-figlio [conta]="locConta"></app-figlio>
```

Prima di tutto con [conta]="locConta" dovete associare la variabile **conta** del componente figlio con la variabile **locConta** del componente padre.

```
<!--figlio.component.html-->
<p>
Vista (figlio)<br>
Conteggio: {{conta}}
</p>
```

La variabile **conta** viene usata nella vista del componente figlio per visualizzare il valore del conteggio.

Binding e Comunicazione

18/43 - @Input

Nel componente figlio, con il decoratore @Input dovete dichiarare la variabile che riceve il valore dal componente padre.

```
//figlio.component.ts
import { Component, Input} from '@angular/core';

@Component({
selector: 'app-figlio',
templateUrl: './figlio.component.html',
styleUrls: ['./figlio.component.css']
})
export class FiglioComponent{
@Input() conta:number;
constructor() { }
}
```

Con il decoratore @Input dichiarate la variabile che riceve il valore dal componente padre.

Binding e Comunicazione

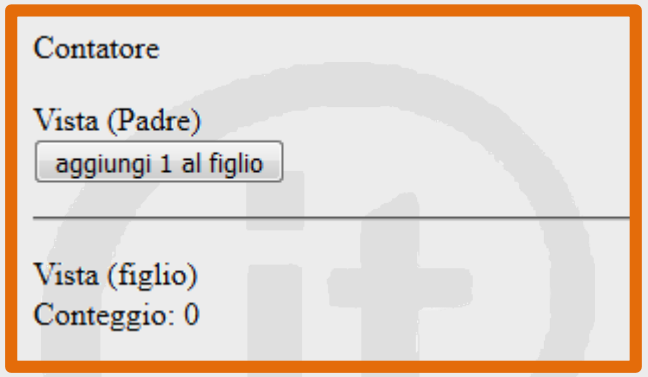
19/43 - @Input

Infine nel componente padre dovete usare la variabile locConta, associata a conta, per gestire il conteggio.

```
//app.component.ts
import { Component } from '@angular/core';
import { FiglioComponent } from './figlio/figlio.component';

@Component({
selector: 'app-root',
templateUrl: './app.component.html',
styleUrls: ['./app.component.css']
})
export class AppComponent {
titolo = 'Contatore';
locConta:number=0;
incrementaContatore(){
this.locConta++;
}}
```

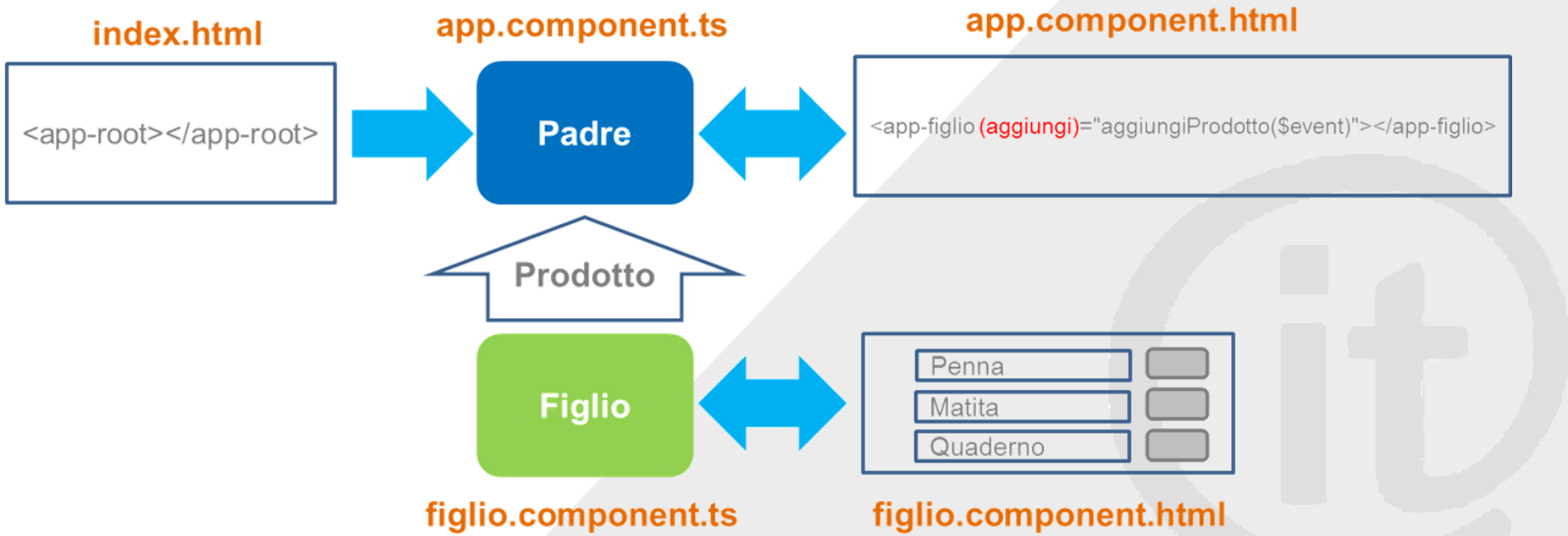
Output



Binding e Comunicazione

20/43 - @Output

Il decoratore **@Output** può essere usato per passare dati da un componente figlio ad un componente padre, come mostra lo schema seguente:



Questa operazione richiede la creazione di un evento da associare al decoratore **@Output**.
L'utilità dell'evento è quella passare le informazioni al componente padre attraverso l'oggetto **$event**.

Binding e Comunicazione

21/43 - @Output

Il decoratore **@Output** può essere usato per passare dati da un componente figlio ad un componente padre, come mostra lo schema seguente:

```
<!— app-component.html -->
Carrello(Padre)<br>
{{prodotto+', '+prezzo}}
<hr>
<app-figlio (aggiungi)="aggiungiProdotto($event)"></app-figlio>
```

Evento personalizzato

Contiene i dati trasmessi del componente figlio

```
<!— app-figlio-component.html -->
Elenco prodotti(Figlio)
<table>
<tr><td>{{prodotti[0].prodotto+', '+prodotti[0].prezzo}}</td><td><button(click)="aggiungiCarrello(0)">aggiungi</button></td></tr>
<tr><td>{{prodotti[1].prodotto+', '+prodotti[1].prezzo}}</td><td><button (click)="aggiungiCarrello(1)">aggiungi</button></td></tr>
<tr><td>{{prodotti[2].prodotto+', '+prodotti[2].prezzo}}</td><td><button (click)="aggiungiCarrello(2)">aggiungi</button></td></tr>
</table>
<button (click)="aggiungiCarrello(3)">Azzera carrello</button>
```

autore Gino Visciano

Binding e Comunicazione

22/43 - @Output

Il decoratore **@Output** può essere usato per passare dati da un componente figlio ad un componente padre, come mostra lo schema seguente:

```
import { Component, Output, EventEmitter} from '@angular/core';
@Component({
selector: 'app-figlio',
templateUrl: './app-figlio.component.html',
styleUrls: ['./app-figlio.component.css']})
export class AppFiglioComponent {
prodotti=[];
indice:number=0;
@Output() aggiungi=new EventEmitter();
constructor() {
this.prodotti.push({prodotto:"Penna",prezzo:1.5});
this.prodotti.push({prodotto:"Matita",prezzo:0.5});
this.prodotti.push({prodotto:"Quaderno",prezzo:2.5});
this.prodotti.push({prodotto:"-",prezzo:0}); //Azzera carrello}
aggiungiCarrello(indice:number){
this.indice=indice;
this.aggiungi.emit(this.prodotti[indice]);}}
```

autore Gino Visciano

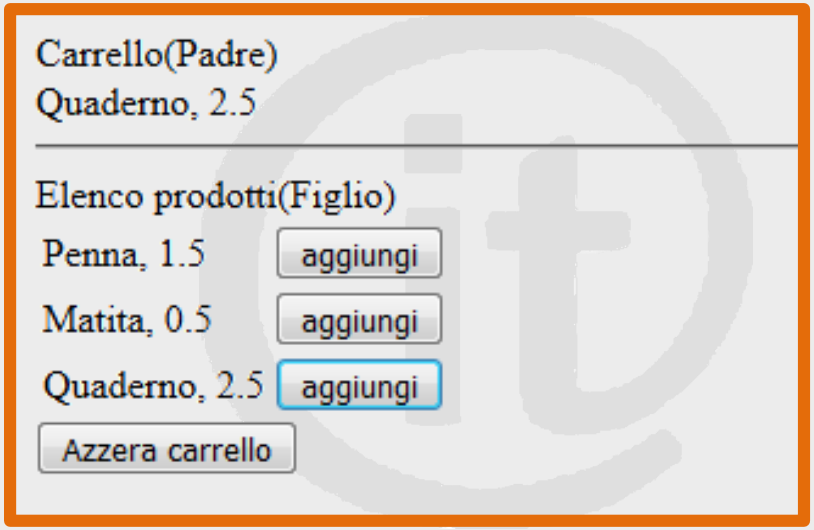
Binding e Comunicazione

23/43 - @Output

Il decoratore **@Output** può essere usato per passare dati da un componente figlio ad un componente padre, come mostra lo schema seguente:

```
import { Component } from '@angular/core';
@Component({
selector: 'app-root',
templateUrl: './app.component.html',
styleUrls: ['./app.component.css']
})
export class AppComponent {
title = 'app';
prodotto:string="-";
prezzo:number=0;
aggiungiProdotto(evento){
this.prodotto=evento.prodotto;
this.prezzo=evento.prezzo;}}
```

Output



autore Gino Visciano

Ciclo di vita di un'Applicazione Angular

24/43 - Ciclo di Hook

In un'applicazione il ciclo di vita dei componenti è gestito da **Angular** e viene chiamato **ciclo di Hook**.
Per gestire il ciclo di vita di un componente in un'applicazione Angular ci sono diverse interfacce:

- OnInit
- OnChanges
- DoCheck
- AfterContentInit
- AfterContentChecked
- AfterViewInit
- AfterViewChecked
- OnDestroy

Per controllare un **Hook** del ciclo di vita di un componente Angular, dovete importare l'interfaccia corrispondente nel componente e sovrascrivere il metodo associato all'interfaccia. Per conoscere quali sono i nomi dei metodi associati alle interfacce è moto semplice, basta aggiungere il prefisso **ng** al nome dell'interfaccia. Ad esempio il metodo da sovrascrivere per gestire la fase corrispondente all'interfaccia OnInit, sarà il seguente:

```
ngOnInit(){
…
}
```

Ciclo di vita di un'Applicazione Angular

25/43 - Sequenza degli Hook

Per usare correttamente i metodi associati alle interfacce del ciclo di Hook, è importante conoscere la sequenza degli Hook. L'elenco seguente mostra l'ordine degli Hook con una breve descrizione:

1. OnChanges
   - ✓ Angular chiama l'Hook OnChanges ogni volta che rileva le modifiche alle proprietà di input del componente (o della direttiva).
2. OnInit
   - ✓ Chiamato una volta sola, dopo che Angular ha finito di creare il componente ed aver eseguito il primo OnChanges. Parte dopo il costruttore.
3. DoCheck
   - ✓ Chiamato ogni volta che si verifica un ciclo di cambiamento non gestito da Angular. Di solito non viene usato, perché OnChanges permette di controllare quasi tutte le modifiche.
4. AfterContentInit/ AfterContentChecked
   - ✓ Chiamati dopo che Angular importa contenuto esterno nel codice Html, ad esempio come avviene per i template.
5. AfterViewInit/AfterViewChecked
   - ✓ Chiamati dopo la creazione di una view o di una fase di rendering.
6. OnDestroy
   - ✓ Chiamato *poco prima* Angular distrugge la direttiva / componente.

autore Gino Visciano

Dipendence Injection

26/43 - Che cos'è la Dipendence Injection

Con **Angular** è possibile iniettare in un componente i riferimenti di un servizio attraverso il **costruttore**, senza doverlo istanziare. In questo modo il servizio può essere condiviso anche tra più componenti perché non viene accoppiato ad uno specifico componente, ma appartiene al **contesto dell'applicazione**.

Per applicare la **Dipendence Injection**, nella classe da iniettare bisogna importare il decoratore seguente:

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

Successivamente il decoratore deve essere inserito prima della classe, come mostra l'esempio seguente:

```
@Injectable()
export class DipendentidaoService {
…
}
```

Infine per associare la **classe** al contesto dell'applicazione, bisogna importarla nel modulo app.module.ts ed aggiungerla nella sezione **providers**.

autore Gino Visciano

Dipendence Injection

27/43 - Come si crea un Servizio

L'esempio seguente mostra i passaggi fondamentali per creare il servizio **DipendentidaoService**,nel modulo dipendentidao.service.ts: Prima di tutto bisogna creare la classe con le funzionalità del servizio ed indicare che dovrà essere iniettata:

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

@Injectable()
export class DipendentidaoService {
// Inserire i metodi di crud per gestire un vettore di tipo Dipendente
}
```

Importazione decoratore Injectable

Decoratore Injectable

Successivamente bisogna aggiungere il servizio al modulo app.module.ts:

```
import { DipendentidaoService } from './dipendentidao.service';

@NgModule({
...,
providers: [DipendentidaoService],
bootstrap: [AppComponent]})
export class AppModule { }
```

Importazione del servizio

Associazione del servizio al contesto applicativo

autore Gino Visciano

Dipendence Injection

28/43 - Come si crea un Servizio

Infine si deve importare il servizio nel componete e richiamarlo nel costruttore come attributo :

```
import { Dipendente } from './Dipendente';
import { DipendentidaoService } from './dipendentidao.service';

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

dipEliminato:number=0;
dips: Dipendente[] = [];
indice: number = 1;
mioForm: FormGroup;
attivaIns:boolean=false;
count:number=0;
idMod:number=0;

constructor(private mioFormBuilder: FormBuilder, private dipService: DipendentidaoService) { }
```

Importazione del servizio

Iniezione del servizio attraverso il costruttore, in questo caso dipService diventa un attributo privato della classe.

Per generare il servizio automaticamente potete usare il comando seguente:

ng generate service dipendentidao

autore Gino Visciano

Form Template Driven

29/43 – ngForm e ngModel

I **Forms Template Driven** vengono utilizzati in **Angular** per creare schede personalizzate gestite principalmente con controlli all'interno della vista.
La direttiva **ngForm** permette di assegnare un riferimento al **Form** per poterlo controllare anche attraverso il componente associato. Per controllare anche i tag input dovete usare la direttiva **ngModel**. Nell'esempio seguente la funzione **onSubmit(f)**, associata alla direttiva **ngSubmit** permette di passare al componente il riferimento del **Form** quando si clicca sul bottone salva.

```
Persona:<br>
<form #f="ngForm" (ngSubmit)="onSubmit(f)" novalidate>
<input name="nome" ngModel required #nome="ngModel"><br>
<input name="cognome" ngModel required #cognome="ngModel"><br>
<input name="eta" ngModel required #eta="ngModel"><br>
<button>Salva</button>
</form>
<p>Nome: {{ nome.value }}</p>
<p>Cognome: {{ cognome.value }}</p>
<p>Età: {{ eta.value }}</p>
<p>Form: {{ f.value | json }}</p>
<p>Form valid: {{ f.valid }}</p>
```

La parola chiave required indica che il campo è obbligatorio

Aggiornamento variabili in tempo reale

True se i valori inseriti nel Form sono validi altrimenti False

autore Gino Visciano

Form Template Driven

30/43 – ngForm e ngModel

Importando la classe **NgForm** nel componete associato alla vista che contiene il **Forms Template Driven**, attraverso il riferimento passato dalla funzione **onSubmit(f:NgForm)** si possono controllare gli elementi a cui è stato assegnato un riferimento di tipo **ngModel**.

Nell'esempio la classe **AppComponent**, riceve il riferimento del **Form** e attraverso il percorso **f.control.get(riferimentoTag)** e visualizza i valori inseriti nella scheda.

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

@Component({
selector: 'app-root',
templateUrl: './app.component.html',
styleUrls: ['./app.component.css']
})
export class AppComponent {
onSubmit(f: NgForm) {
alert(f.control.get('nome').value+', '+f.control.get('cognome').value+', '+f.control.get('eta').value);
alert(f.valid);}}
```

autore Gino Visciano

Form Template Driven

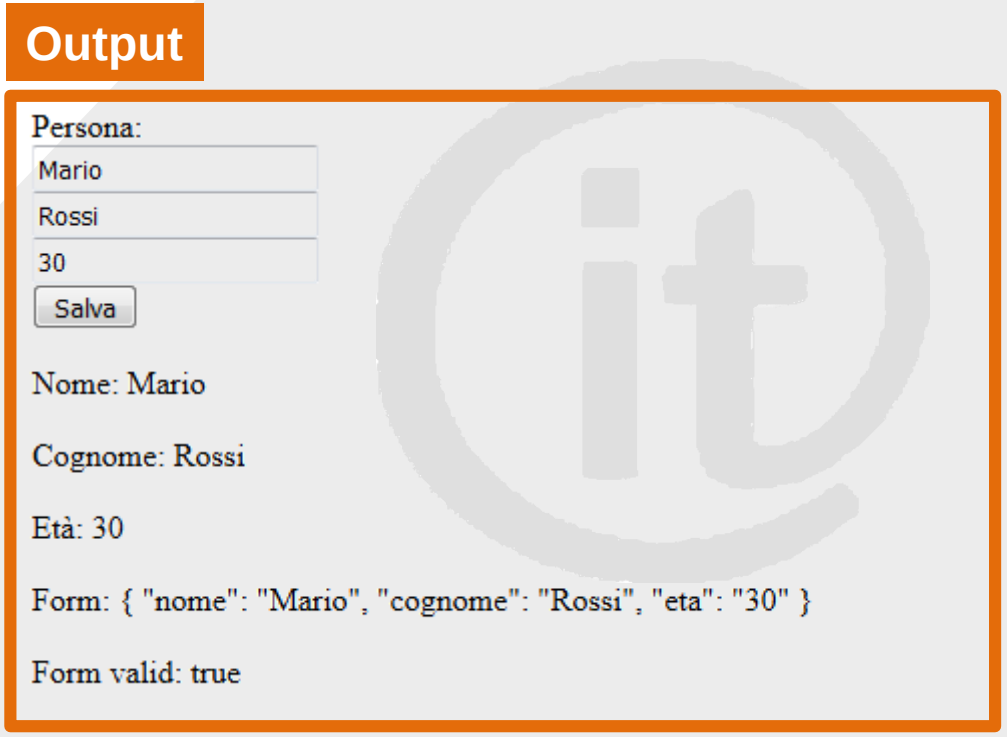
31/43 – ngForm e ngModel

Per utilizzare i **Forms Template Driven**, in **app.module.ts** dovete importare il modulo **FormsModule**.

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

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

Output



autore Gino Visciano

Form Reactive

32/43 – FormGroup e FormControl

I **Forms Reactive** vengono utilizzati in **Angular** per creare schede personalizzate gestite principalmente attraverso i componenti associati alle viste.
Per assegnare un riferimento ad un **Form Reactive** si usa la direttiva **[formGroup]**, per il controllo dei **tag input** si utilizza la parola chiave **formControlName**.

1/2

```
//app-dip-form.component_01.html
<form [formGroup]="mioForm">
<div class="container">
<h2>Dipendente</h2>
<label class="center-block">
Id:<input class="form-control" formControlName="id"></label><br>
<label class="center-block">
Nome:<input type="text" class="form-control" formControlName="nome">
<span *ngIf="mioForm.controls.nome.dirty && mioForm.controls.nome.errors">Nome
obbligatorio</span></label><br>
<label class="center-block">
Cognome:<input class="form-control" formControlName="cognome">
<span *ngIf="mioForm.controls.cognome.dirty && mioForm.controls.cognome.errors">Cognome
obbligatorio</span></label><br>
```

continua …

autore Gino Visciano

Form Reactive

33/43 – FormGroup e FormControl

```
<label class="center-block">Età:<input class="form-control" formControlName="eta">
<span *ngIf="mioForm.controls.eta.dirty &&
!mioForm.controls.eta.valid">{{mioForm.controls.eta.errors.nonValida}}</span>
</label><br>
<label class="center-block">Stipendio:<input class="form-control" formControlName="stipendio">
</label><br>
<button (click)="salva()">Salva</button>
</div>
</form>
<pre>{{mioForm.value | json}}</pre>
<p *ngFor="let dip of dips">{{dip | json}}</p>
```

2/2

fine

Il **tag span** seguente visualizza il messaggio d'errore **'Nome obbligatorio'** se il nome non viene inserito dopo la modifica del valore originario del campo (**dirty**).

```
<span *ngIf="mioForm.controls.nome.dirty && mioForm.controls.nome.errors">Nome obbligatorio</span>
```

Form Reactive

34/43 – FormGroup e FormControl

Di seguito sono indicati gli **stati** che possono assumere i campi di un **Form Reactive**.

Gli stati servono per gestire la **validazione dei dati**.

- **Touched e Untouched**

Questi due stati indicano se un elemento di input è stato visitato dall'utente, cioè se almeno una volta il campo ha perso il **focus**. Se l'elemento non ha mai perso il focus la sua proprietà **touched** ha valore **false**, mentre **untouched** è **true**; in caso contrario i valori si invertono.

- **Dirty e Pristine**

Questi due stati indicano se il valore di un campo è stato modificato rispetto al valore originario. Se non è stato modificato, il valore della proprietà **dirty** è **false** mentre quello della proprietà **pristine** è **true**; i valori si invertono nella situazione opposta.

- **Valid e Invalid**

Queste proprietà indicano se i valori presenti nella form sono validi secondo i criteri di validazione impostati. Come è naturale, **valid** e **invalid** prendono il valore booleano corrispondente nel caso in cui un campo sia valido o meno.

autore Gino Visciano

Form Reactive

35/43 – FormGroup e FormControl

Per collegare un **Form Reactive** creato nella **vista** con il **componente** associato, dovete procedere come segue:

1. importare le classi: **FormGroup** e **FormControl**
2. creare all'interno del metodo **ngOnInit** un oggetto di tipo **FormGroup**
3. passare al costruttore **FormGroup(…)** un oggetto con attributi di tipo **FormControl**, con lo stesso nome assegnato ai riferimenti di tipo **formControlName** nella vista, come mostra l'esempio seguente:

```
// app-dip-form.component.ts
import { Component, OnInit } from '@angular/core';
import { FormGroup, FormControl, Validators, AbstractControl} from '@angular/forms';
import { Dipendente } from '../dipendente';
@Component({
selector: 'app-dip-form1',
templateUrl: './app-dip-form.component_01.html',
styleUrls: ['./app-dip-form.component.css']})
export class AppDipFormComponent implements OnInit {
dips:Dipendente[]=[];
indice:number=1;
```

1/3

continua …

autore Gino Visciano

Form Reactive

36/43 – FormGroup e FormControl

```
mioForm:FormGroup;
constructor() {}
ngOnInit() {
this.mioForm=new FormGroup({
id : new FormControl({value:this.indice,disabled:true}),
nome : new FormControl('',[Validators.required,Validators.maxLength(15)]),
cognome : new FormControl('',Validators.required),
eta : new FormControl('',Validators.compose([this.validaEta])),
sesso : new FormControl(),
stipendio : new FormControl()});}
salva(){
if(this.mioForm.valid){
let dip:Dipendente=new
Dipendente(this.indice,this.mioForm.get("nome").value,this.mioForm.get("cognome").value,
this.mioForm.get("eta").value,true,this.mioForm.get("stipendio").value);
this.dips.push(dip);
this.mioForm.get("id").setValue(++this.indice);
}}
```

Dichiarazione riferimento di tipo FormGroup con lo stesso nome assegnato al Form nella vista.

Istanza dell'oggetto FormGroup e degli attributi FormControl, con lo stesso nome assegnato ai riferimenti FormControlName nella vista.

2/3

continua …

Form Reactive

37/43 – FormGroup e FormControl

Per validare il contenuto di un **Form Reactive** dovete importare le classi **Validators** e **AbstractControl**. L'esempio seguente mostra come utilizzare la classe Validators nel componente:

```
nome : new FormControl('',[Validators.required,Validators.maxLength(15)]),
cognome : new FormControl('',Validators.required),
```

Potete anche aggiungere controlli di validazione personalizzati attraverso la creazione di funzioni di validazione, come mostra l'esempio seguente:

```
validaEta(control : AbstractControl) : {[key : string]:any;} {
if(control.value.length==0){
return {'nonValida': "L'età è obbligatoria"}}
var isValid = /[0-9]+/.test(control.value);
if (isValid) {
return null;}
return {'nonValida': "L'età deve essere un numero"}}}
```

La funzione riceve il riferimento di tipo AbstractControl del controllo a cui è stata collegata e ritorna un oggetto con una chiave stringa uguale ad un informazione di qualunque tipo (any).

3/3

fine

Per collegare la funzione di validazione personalizzata al controllo dovete usare il metodo **Validators.compose**, come mostra l'esempio seguente:

```
eta : new FormControl('',Validators.compose([this.validaEta]))
```

Form Reactive

38/43 – FormGroup e FormBuilder

Il servizio **FormBuilder** può essere usato per rendere ancora più semplice il collegamento tra gli attributi del FormGroup ed i rifermenti di tipo **formControlName** nella vista. Per usare questo servizio lo dovete iniettare nel componente attraverso il costruttore, come mostra l'esempio:

```
//app-dip-form.component_02.ts
import { Component, OnInit } from '@angular/core';
import { FormGroup, FormControl, FormBuilder,Validators, AbstractControl} from '@angular/forms';
import { Dipendente } from '../dipendente';
@Component({
selector: 'app-dip-form2',
templateUrl: './app-dip-form.component_02.html',
styleUrls: ['./app-dip-form.component.css']})
export class AppDipFormComponent implements OnInit {
dips:Dipendente[]=[];
indice:number=1;
mioForm:FormGroup;
constructor(private mioFormBuilder: FormBuilder) {}
```

1/2

Iniezione del Servizio FormBuilder nel componente attraverso il costruttore.

continua …

autore Gino Visciano

Form Reactive

39/43 – FormGroup e FormBuilder

```
ngOnInit() {
this.mioForm=this.mioFormBuilder.group({
id : [{value:this.indice,disabled:true}],
nome : ['',Validators.compose([Validators.required,Validators.maxLength(15)])],
cognome : ['',Validators.required],
eta : ['',Validators.compose([this.validaEta])],
sesso : true,
stipendio :''});}
salva(){
if(this.mioForm.valid){
let dip:Dipendente=new Dipendente(this.indice,this.mioForm.get("nome").value,this.mioForm.get("cognome")
.value,this.mioForm.get("eta").value,true,this.mioForm.get("stipendio").value);
this.dips.push(dip);
this.mioForm.get("id").setValue(++this.indice);}}
validaEta(control : AbstractControl) : {[key : string]:any;} {
if(control.value.length==0){return {'nonValida': "L'età è obbligatoria"}}
var isValid = /[0-9]+/.test(control.value);
if (isValid) {return null;}
return {'nonValida': "L'età deve essere un numero"}}}
```

Collegamento del Servizio FormBuilder al FormGroup.

Grazie al FormBuilder potete dichiarare direttamente gli attributi del FormGroup senza usare il FormControl.

2/2

fine

autore Gino Visciano

Form Reactive

40/43 – FormGroup e FormArray

All'interno di un **Form Reactive** è possibile aggiungere vettori di **FormControl** utilizzando la classe **FormArray**. L'esempio seguente mostra come si gestisce questo tipo di controllo senza utilizzare il servizio **FormBuilder**

```
import { FormGroup, FormControl, FormArray,Validators, ValidatorFn, AbstractControl} from '@angular/forms';
…
ngOnInit() {
this.mioForm=new FormGroup({
id: new FormControl({value:this.indice, disabled:true}),
nome: new FormControl(),
cognome: new FormControl(),
eta: new FormControl(),
sesso: new FormControl(),
stipendio: new FormControl(),
mails: new FormArray([])});
…
aggiungiControllo(m:string){
this.mails.push(new FormControl(m));
}
eliminaControllo(i){
this.mails.removeAt(i);
}
```

Vettore di FormControl.

Aggiunge un nuovo controllo al vettore di controlli.

Rimuove un controllo dal vettore di controlli.

autore Gino Visciano

Form Reactive

41/43 – FormGroup e FormArray

L'esempio seguente mostra come si gestisce il controllo **FormArray** utilizzando il servizio **FormBuilder**:

```
import { FormGroup, FormControl, FormArray,Validators, ValidatorFn, AbstractControl} from '@angular/forms';
…
constructor(private mioFormBuilder: FormBuilder) {}
ngOnInit() {
this.mioForm=this.mioFormBuilder.group({
id: {value:this.indice, disabled:true},
nome:['',[Validators.required,Validators.maxLength(15)]],
cognome:['',[Validators.required,Validators.maxLength(20)]],
eta:['',Validators.compose([this.validaEta])],
sesso:['sesso',Validators.compose([this.validaSesso])],
stipendio:'0.00',
mails: this.mioFormBuilder.array([])});
…
aggiungiControllo(m:string){
this.mails.push(new FormControl(m));
}
eliminaControllo(i){
this.mails.removeAt(i);
}
```

Vettore di FormControl.

Aggiunge un nuovo controllo al vettore di controlli.

Rimuove un controllo dal vettore di controlli.

Form Reactive

42/43 – FormGroup e FormArray

Nelle **view** per visualizzare i controlli aggiunti ad un **FormArray** dovete impostare una sezione utilizzando un **tag div** con l'attributo **formArrayName**. All'attributo dovete assegnare il nome del **FormArray** creato nel componente. Infine per visualizzare i controlli dovete usare una direttiva **ngFor**, come mostra l'esempio seguente:

```
M A I L
<div formArrayName="mails">
<div *ngFor="let mail of mails.controls; let i = index">
<input type="text" formControlName="{{i}}"><button (click)="eliminaControllo(i)"> - </button>
</div>
</div>
<button type="button" (click)="aggiungiControllo()"> AGGIUNGI MAIL </button><br>
<button type="submit"> S A L V A </button><br>
</form>
<pre>{{mioForm.value | json}}</pre>
<p *ngFor="let dip of dips">{{dip | json}}</p>
```

Per collegare la sezione impostata con il **tag div** e l'attributo **formArrayName** con il componente dovete utilizzare il comando seguente:

```
get mails(): FormArray { return this.mioForm.get('mails') as FormArray; }
```

autore Gino Visciano

Form Reactive

43/43 – FormGroup e FormArray

Output

Form Builder Dipendente

Id: 1
Nome:
Cognome:
Età:
Sesso: --- Sesso --- Scegli il sesso
Stipendio: 0.00
M A I L
a@a -
b@b -
c@c -
AGGIUNGI MAIL
S A L V A

```
{
  "nome": "",
  "cognome": "",
  "eta": "",
  "sesso": "sesso",
  "stipendio": "0.00",
  "mails": [
    "a@a",
    "b@b",
    "c@c"
  ]
}
```

Vettore di FormControl.


THANK YOU
for watching
www.italdata.it