



UNIVERSITÀ
DEGLI STUDI
DI PALERMO



ANGULAR

Corso Programmazione Web & Mobile

A.A 2024/2025

ING. LUCA CRUCIATA

Angular

Che cos'è Angular?

- Angular è una piattaforma e un framework per la creazione di applicazioni web lato client.
- Sviluppato e gestito da Google, basato su **TypeScript** e supporta HTML e **CSS** per il templating e lo styling.

Caratteristiche principali di Angular

- Architettura basata su componenti:
- Le applicazioni sono costituite da componenti riutilizzabili.
- I componenti controllano una parte dell'interfaccia utente (UI).

Legame dei dati a due vie:

- Sincronizza i dati tra il modello (logica) e la vista (interfaccia utente) in tempo reale.

Pre-requisiti

Node.js è un ambiente di runtime che consente di eseguire codice JavaScript lato server. Include **npm (Node Package Manager)**, un gestore di pacchetti essenziale per installare librerie e dipendenze necessarie nello sviluppo di applicazioni web.

Sebbene Angular sia un framework front-end, l'uso di Node.js è fondamentale durante lo sviluppo per diversi motivi:

- **Gestione delle dipendenze:** npm, fornito con Node.js, permette di installare e gestire le librerie necessarie per Angular, inclusa l'installazione di Angular CLI.
- **Server di sviluppo locale:** Durante lo sviluppo, Angular utilizza un server locale basato su Node.js per testare l'applicazione in tempo reale.
- **Compilazione e build:** Strumenti come Webpack e TypeScript, utilizzati per compilare e ottimizzare il codice Angular, funzionano in ambiente Node.js.

Installazione da Node.js - v[^][18.19.1 or newer](#) (versione LTS Long Term Support)

Installazione

-Installazione di node richiesta:

```
node --version
```

-Installazione di npm necessaria ma su Windows viene fatto in automatico, con node.

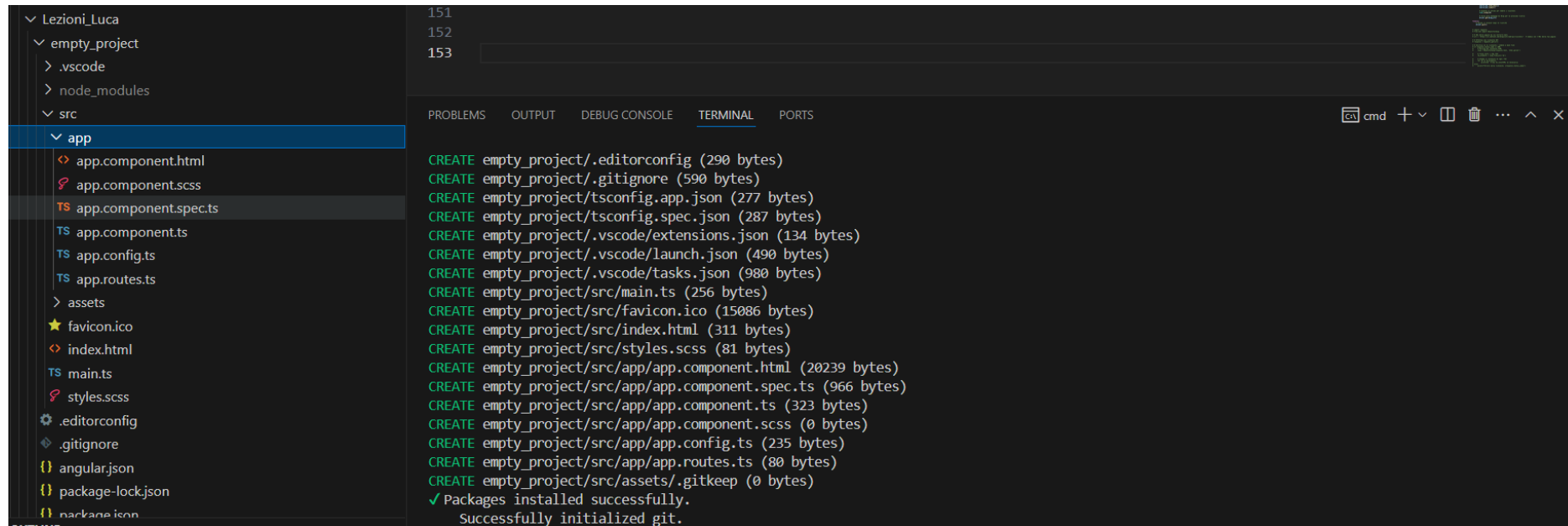
-Installazione di angular

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

Creare un nuovo progetto

Utilizzando la CLI possiamo creare un nuovo progetto

`ng new nome_progetto`



The screenshot shows the Visual Studio Code interface. On the left, the File Explorer displays the project structure for 'empty_project', including folders like '.vscode', 'node_modules', 'src', and 'app'. The 'app' folder is expanded, showing files like 'app.component.html', 'app.component.spec.ts', 'app.component.ts', 'app.config.ts', 'app.routes.ts', 'assets', 'favicon.ico', 'index.html', 'main.ts', 'styles.scss', '.editorconfig', '.gitignore', 'angular.json', 'package-lock.json', and 'package.json'. The right side of the image shows the TERMINAL panel with the following output:

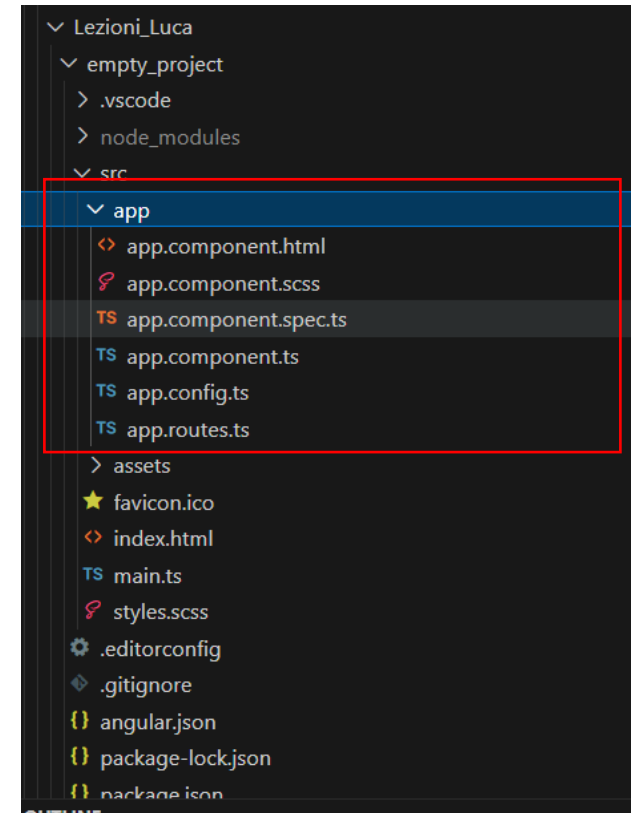
```
151  
152  
153  
CREATE empty_project/.editorconfig (290 bytes)  
CREATE empty_project/.gitignore (590 bytes)  
CREATE empty_project/tsconfig.app.json (277 bytes)  
CREATE empty_project/tsconfig.spec.json (287 bytes)  
CREATE empty_project/.vscode/extensions.json (134 bytes)  
CREATE empty_project/.vscode/launch.json (490 bytes)  
CREATE empty_project/.vscode/tasks.json (980 bytes)  
CREATE empty_project/src/main.ts (256 bytes)  
CREATE empty_project/src/favicon.ico (15086 bytes)  
CREATE empty_project/src/index.html (311 bytes)  
CREATE empty_project/src/styles.scss (81 bytes)  
CREATE empty_project/src/app/app.component.html (20239 bytes)  
CREATE empty_project/src/app/app.component.spec.ts (966 bytes)  
CREATE empty_project/src/app/app.component.ts (323 bytes)  
CREATE empty_project/src/app/app.component.scss (0 bytes)  
CREATE empty_project/src/app/app.config.ts (235 bytes)  
CREATE empty_project/src/app/app.routes.ts (80 bytes)  
CREATE empty_project/src/assets/.gitkeep (0 bytes)  
✓ Packages installed successfully.  
Successfully initialized git.
```

Nel esempio abbiamo generato un progetto chiamato empty_project

Creare un nuovo progetto

In automatico viene generato il AppComponent serve da container dell'intera applicazione, con i file predefiniti:

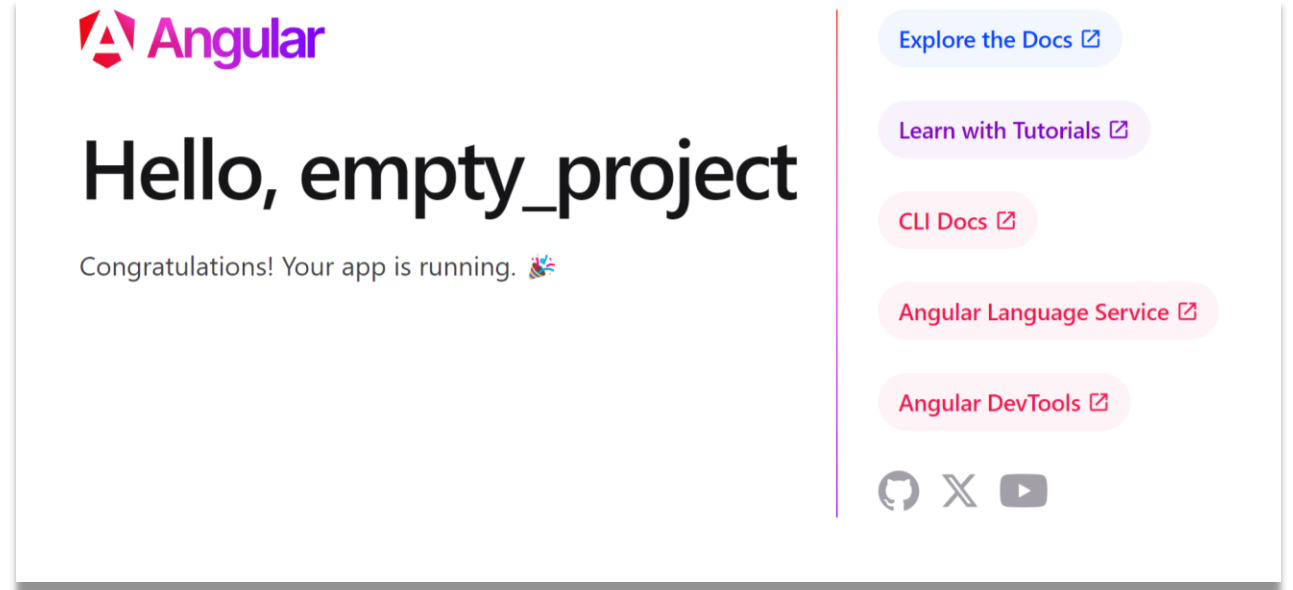
- *app.html*
- *app.ts*
- *app.scss*



Eseguire il progetto

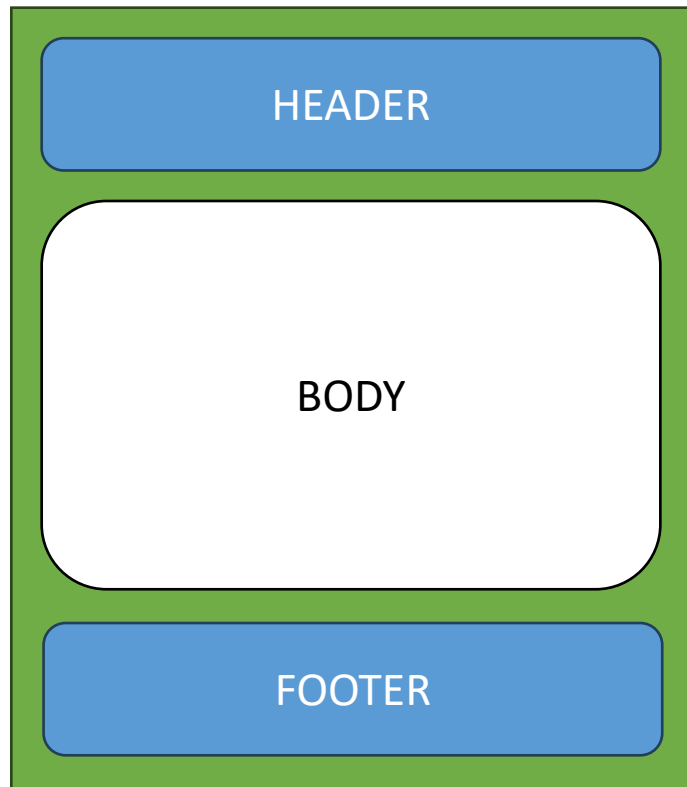
Per eseguire l'applicazione dobbiamo eseguire da terminale *ng serve* oppure *npm start* (*eseguire nella directory del progetto*)

Con l'ulteriore opzione *- -open* verrà aperta l'applicazione, in alternativa possiamo raggiungere il dominio con *localhost:4200*



Componenti

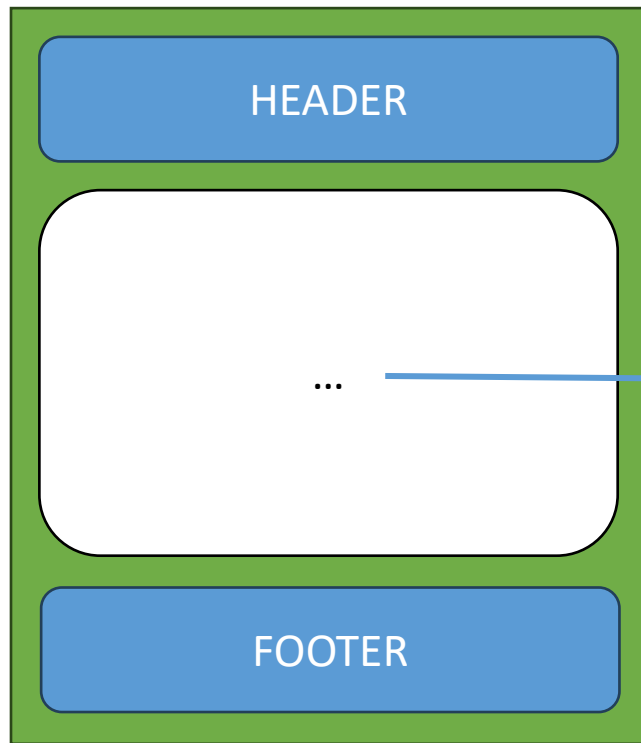
Sono gli elementi base che permettono di rendere modulari le app.



Nell'esempio in figura, il verde è l'AppComponent che funge da padre, mentre gli altri sono HeaderComponent, BodyComponent e FooterComponent che sono figli.

Componenti

Per enfatizzare il concetto di modularità e mostrare l'utilità di un approccio component-based.



Posso inserire componenti diversi, per ottenere una struttura statica con comportamento dinamico.

Un approccio di questo tipo offre diversi vantaggi e permette:

- riduce al minimo la duplicazione del codice.
- favorisce la manutenzione del codice.
- alto livello di riusabilità.
- Favorisce il testing del codice.

Creazione di un nuovo componente

Comando da CLI, da eseguire nel path `./src/app`

ng generate component nome_componente

- TypeScript class
- HTML template
- CSS styles

```
1  import { Component } from '@angular/core';
2
3  @Component({
4    selector: 'app-home',
5    templateUrl: './home.component.html',
6    styleUrls: ['./home.component.scss']
7  })
8  export class HomeComponent {
9
10 }
```

Selettore/tag con il quale richiamare il componente nella parte HTML.
`<app-home></app-home>`

Riferimento al file html del componente

Riferimento al foglio di stile del componente

Struttura del Componente

```
// user-profile.ts
```

```
@Component({
```

```
  selector: 'user-profile',
```

```
  template: `
```

```
    <h1>User profile</h1>
```

```
    <p>This is the user profile page</p>
```

```
  `;
```

```
})
```

```
export class UserProfile { /* Your  
  component code goes here */ }
```

- A @Component [decorator](#) che contiene le istruzioni usate da *Angular*.
- Il template HTML controlla la renderizzazione nel DOM.
- Una classe TypeScript con comportamenti, come la gestione degli input dell'utente o l'invio di richieste a un server.

Struttura del Componente

```
// user-profile.ts
@Component({
  selector: 'user-profile',
  template: `
    <h1>User profile</h1>
    <p>This is the user profile page</p>
  `;
})
export class UserProfile { /* Your
component code goes here */ }
```

A `@Component` [decorator](#) che contiene le istruzioni usate da *Angular*.

A [CSS selector](#) che definisce come il componente è utilizzato nel template HTML.

Il template HTML controlla la renderizzazione nel DOM.

Una classe TypeScript con comportamenti, come la gestione degli input dell'utente o l'invio di richieste a un server.

Struttura del Componente

```
// user-profile.ts
@Component({
  selector: 'user-profile',
  template: `
    <h1>User profile</h1>
    <p>This is the user profile page</p>
  `,
  styles: `h1 { font-size: 3em; }`,
})
export class UserProfile { /* Your
component code goes here */ }
```

- A [@Component decorator](#) that contains some configuration used by Angular.
- A [CSS selector](#) that defines how the component is used in HTML.
- An HTML template that controls what renders into the DOM.
- A TypeScript class with behaviors, such as handling user input or making requests to a server.
- Possiamo anche definire nel decoratore del Componente, lo stile che vogliamo utilizzare all'interno di esso.

Importare risorse esterne

Per rendere il codice più leggibile e in un'ottica di modularità delle singole parti, cioè:

- HTML, per la parte visuale
- CSS, per lo stile della pagina
- Typescript, per la logica della pagina

Conviene definire file esterni, che importiamo tramite **templateUrl** e **styleUrl**.

```
// user-profile.ts
@Component({
  selector: 'user-profile',
  templateUrl: 'user-profile.html',
  styleUrls: ['user-profile.css'],
})
export class UserProfile {
  // Component behavior is defined in here
}
```

Html file URL

```
<!-- user-profile.html -->
<h1>Use profile</h1>
<p>This is the user profile page</p>
```

CSS file URL

```
/* user-profile.css */
h1 {
  font-size: 3em;
}
```

Componenti

Come abbiamo visto i componenti vengono definiti come classi quindi possiamo definire delle proprietà e delle funzioni di classe.

```
3  @Component({
4    selector: 'app-home',
5    templateUrl: './home.component.html',
6    styleUrls: ['./home.component.scss']
7  })
8  export class HomeComponent {
9
10     title = "HOME";
11
12     saluta(name: String) {
13       return "Benvenuto" + name;
14     } /* Funzione */
15
16
17 }
```

Logica

home works!

Benvenuto Bob sei nella pagina: HOME

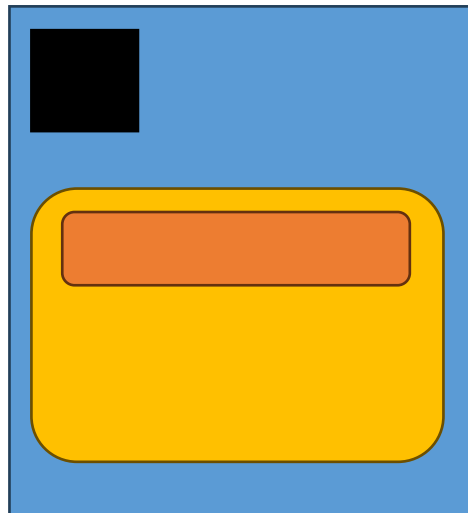
HTML

```
1  <p>home works!</p>
2
3  <p>{{saluta("Bob")}} sei nella pagina: {{title}}</p>
4
```

Logica dei Componenti

Supponiamo di voler costruire un componente UserProfile che al suo interno deve contenere informazioni diverse. Possiamo pensare di definire un componente per la gestione delle singole parti, cioè definiamo comportamenti basici.

Questi saranno utilizzati in un componente, che fungerà da contenitore per gli altri.



Componente ProfilePhoto

UserBio

UserAddressInfo

UserProfile

Importare altri componenti

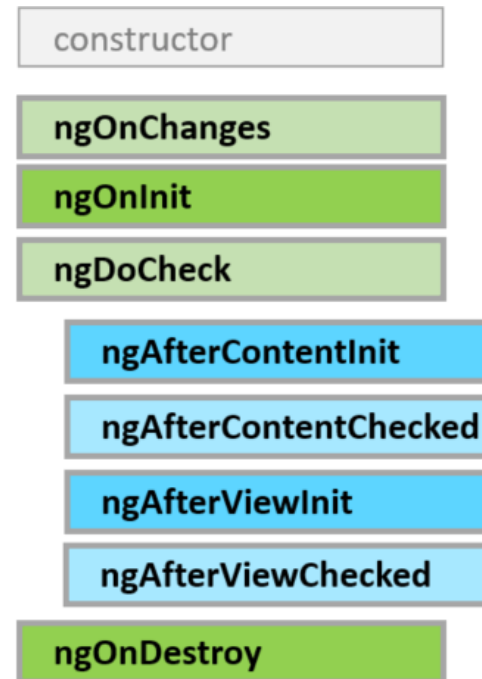
Quindi nel nostro esempio andremo, a importare gli altri componenti nel componente padre.

```
// user-profile.ts
import {ProfilePhoto} from 'profile-photo.ts';
import {UserBio} from 'user-bio.ts';

@Component({
  selector: 'user-profile',
  imports: [ProfilePhoto, UserBio],
  template: `
    <h1>User profile</h1>
    <profile-photo /> <p>This is the user profile page</p> `,
  })
export class UserProfile { // Component behavior is defined in here}
```

Component Lifecycle

I **lifecycle hooks** in Angular sono metodi di Angular che permettono di interagire con le diverse fasi del ciclo di vita di un componente o di una direttiva. Questi metodi offrono la possibilità di gestire eventi come l'inizializzazione, i cambiamenti nei dati, o la distruzione del componente.



Lifecycle

Phase	Method	Summary
Creation	constructor	Costruttore classico, eseguito dopo aver istanziato il componente.
ChangeDetection	ngOnInit	Eseguito dopo aver iniziato tutti gli input al componente.
	ngOnChanges	Eseguito ogni volta che cambiano gli input al componente.
	ngDoCheck	Eseguito per controllare se il componente è stato modificato.
	ngAfterContentInit	Viene eseguito una volta dopo che il contenuto del componente è stato inizializzato.
	ngAfterContentChecked	Viene eseguito ogni volta che il contenuto di questo componente è stato controllato per le modifiche.
	ngAfterViewInit	Viene eseguito una volta dopo l'inizializzazione della vista del componente.
Rendering	ngAfterViewChecked	Runs every time the component's view has been checked for changes.
	afterNextRender	Viene eseguito ogni volta che la vista del componente è stata controllata per le modifiche.
Destruction	afterRender	Viene eseguito ogni volta che tutti i componenti sono stati resi nel DOM.
	ngOnDestroy	Viene eseguito una volta prima che il componente venga distrutto.

ngOnInit

- Il metodo **ngOnInit** viene eseguito dopo che Angular ha inizializzato tutti gli input del componente con i loro valori iniziali. Il metodo **ngOnInit** di un componente viene eseguito esattamente una volta.
- Questo passaggio avviene prima che il template del componente venga inizializzato. Ciò significa che è possibile aggiornare lo stato del componente in base ai suoi valori iniziali di input.
- Utile per inizializzare dati o chiamare API.

Altri cyclehook

- **ngOnChanges** controlla variazioni sugli Input al componente. Utilizzato per monitorare variazioni sui dati ricevuti dal componente genitore.
- **ngOnDestroy** chiamato prima della distruzione di un componente, serve per rilasciare risorse e annullare sottoscrizioni.

Controlli di flusso NgIf vs @If

Angular offre la possibilità di gestire la visualizzazione degli elementi condizionati al loro valore, con un costrutto IF, IF-ELSE per guidare il comportamento. Nell'esempio abbiamo definito la proprietà `logged=true`. ***ngIf** vecchie versioni quasi deprecato, **@If** nuove versioni, ma hanno lo stesso comportamento.

HTML

```
1 <p>{{title}} Component Example</p>
2
3 @if (logged){
4   <p>Sei già loggato &#64;if </p>
5 }
6 @else{
7   <p>Non sei ancora loggato &#64;if</p>
8 }
9
10 <p *ngIf="logged">Sei già loggato *NgIf</p>
11 <p *ngIf="!logged">Non sei ancora loggato *NgIf</p>
```

Cosa visualizziamo

IF control-flow Component Example

Sei già loggato @if

Sei già loggato *NgIf

@if @else

Perché non dobbiamo importare @if?

Si noti che non è necessario importare il @if direttiva da @angular/common nei nostri modelli di componenti più.

Questo perché il @if la sintassi fa parte del motore modello stesso, ed è **non** una direttiva.

Il nuovo @if è integrato direttamente nel motore modello, quindi è automaticamente disponibile ovunque.

Perché è il nuovo @if sintassi migliore di *ngIf?

Riassumiamo rapidamente i motivi principali per cui @if è meglio di *ngIf:

- meno prolisso, più intuitivo
- nessuna necessità di importazioni
- supporti else if e else condizioni
- nessun sovraccarico di runtime
- renderà più semplice la futura evoluzione del quadro

Controlli di flusso NgFor vs @For

Angular offre la possibilità di iterare su alcuni elementi iterabili e creare elementi HTML associati ad essi.

Supponiamo di avere questa lista:

```
people = [  
  { name: 'Alice', age: 28, profession: 'Engineer' },  
  { name: 'Bob', age: 34, profession: 'Designer' },  
  { name: 'Charlie', age: 25, profession: 'Developer' },  
];
```

```
3 @for (p of people; track p.name) {  
4   {{ p.name }}  
5 }
```

Alice Bob Charlie

```
7 @for (p of people; track p.name) {  
8   <p>{{ p.name }}</p>  
9 }
```

Alice
Bob
Charlie

L'uso di **track** è richiesto, necessita un identificatore unico.

Controlli di flusso NgFor vs @For

```
people = [  
  { name: 'Alice', age: 28, profession: 'Engineer' },  
  { name: 'Bob', age: 34, profession: 'Designer' },  
  { name: 'Charlie', age: 25, profession: 'Developer' },  
];
```

```
11 <ul>  
12   @for (p of people; track p.name) {  
13     <li>{{ p.name }}</li>  
14   }  
15 </ul>
```

- Alice
- Bob
- Charlie

```
17 <p *ngFor="let p of people">{{p.name}}</p>
```

Alice
Bob
Charlie

Binding

Il **binding** in Angular è un meccanismo che consente di collegare i dati tra il **model** (la logica dell'applicazione) e il **view** (interfaccia utente). Grazie al binding, i dati e le interazioni dell'utente sono sincronizzati in modo automatico, rendendo il codice più semplice e reattivo.

Il binding viene utilizzato per impostare dinamicamente i valori di proprietà e attributi.

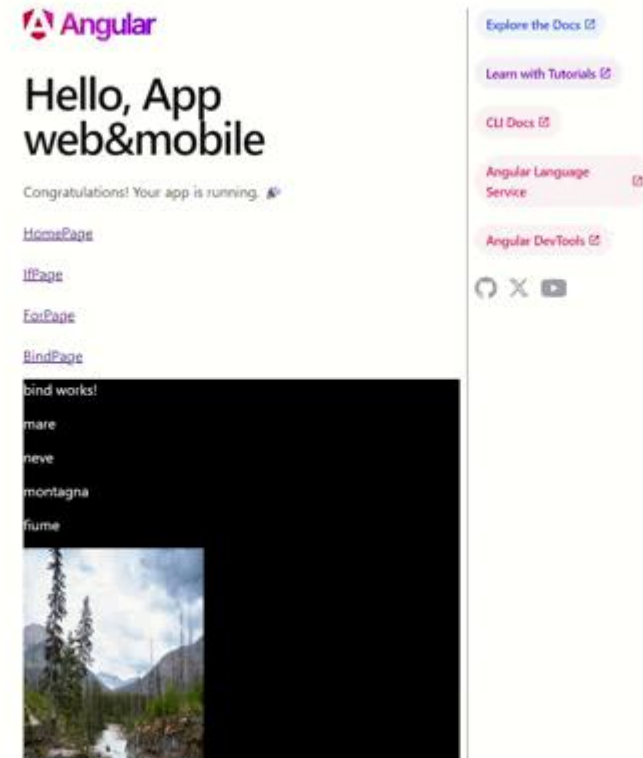
```
3  @for (image of images; track image.type) {
4    <p (click)="select(image.path)" style="cursor: pointer;">
5      {{ image.type }}
6    </p>
7  }
8
9  <img [src]="path" width="200" height="400">

```

```
images= [
  {type:"mare", path:"https://images"},
  {type:"neve", path:"https://plus.u"},
  {type:"montagna", path:"https://p"},
  {type:"fiume", path:"https://plus.

```

```
select(selected_path:string){
  this.path= selected_path;
}
```



Gestione degli eventi

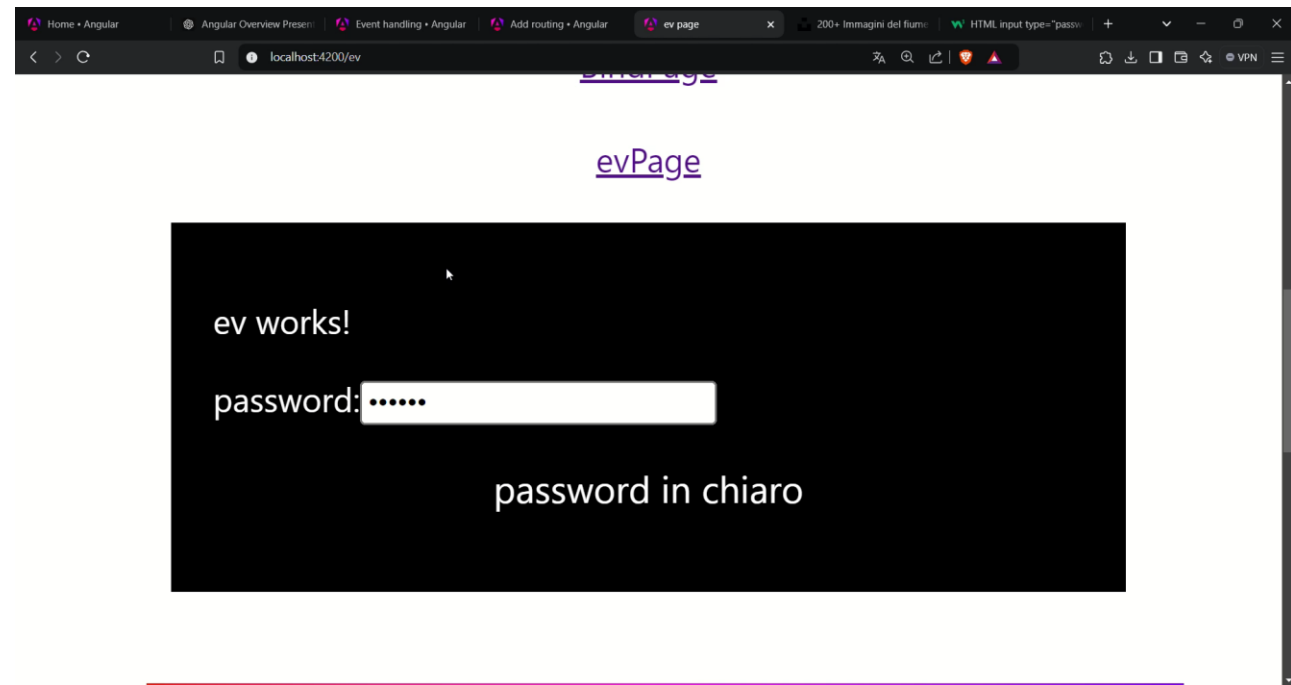
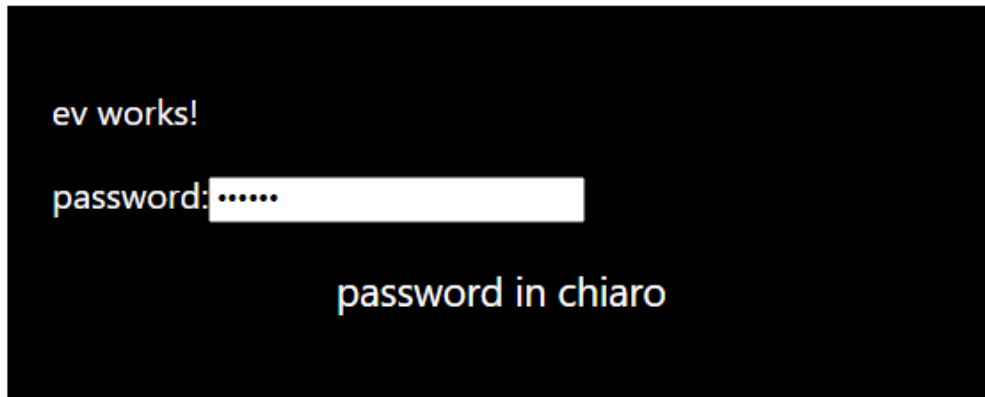
Gestendo gli eventi possiamo monitorare click del mouse o della tastiera, sottomissioni di form, ecc... e decidere come «reagire» a queste azioni dell'utente.

```
3  @for (image of images; track image.type) {  
4      <p (click)="select(image.path)" style="cursor: pointer;">  
5          {{ image.type }}  
6      </p>  
7  }  
8  
9  <img [src]="path" width="200" height="400">
```

Nell'esempio in figura ogni volta che l'utente clicca sul paragrafo richiamiamo la funzione di classe *select(image_path)*

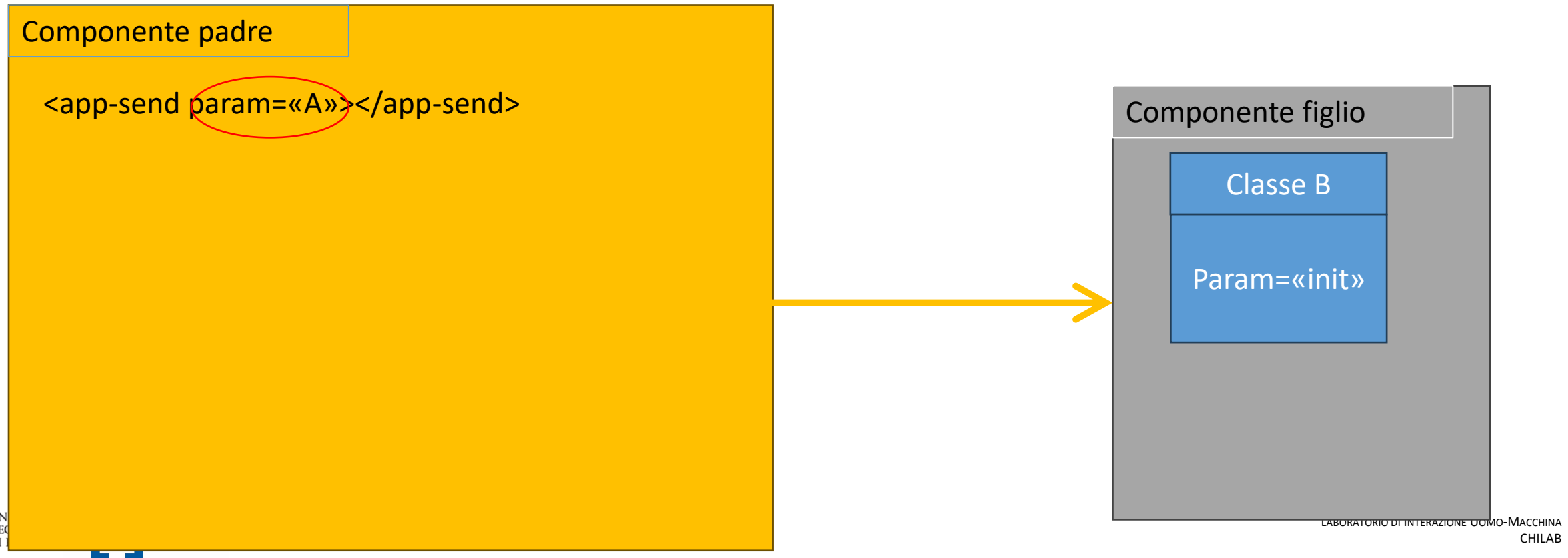
```
select(selected_path:string){  
  this.path= selected_path;  
}
```

Gestione degli eventi



Input ad un componente

Possiamo definire delle proprietà con il decoratore `@Input`, questo permette di iniettare dei valori a tale proprietà da fuori il componente. In particolare in una logica gerarchica padre-figlio possiamo passare i parametri, quando definiamo il selector all'interno del content del padre.



Input ad un componente

```
<app-send parameter="Send Component"></app-send>  
<app-send></app-send>
```

Componente Padre

```
export class SendComponent {  
  @Input() parameter= "vuoto";  
}
```

Componente Figlio

send works!

Argomento ricevuto: Send Component

Risultato con parametro

send works!

Argomento ricevuto: vuoto

Risultato senza parametro

Input ad un componente con `input<T>()`

- Un altro metodo per la gestione degli input, viene realizzato con la funzione `input`

snippet di codice

Il valore passato sarà considerato come Input Signal ma richiamando il parametro come funzione potremo accedere direttamente al suo valore.

@Output e EventEmitter

I dati passati al componente figlio possono essere elaborati, visualizzati, ecc...

Capita che dopo l'elaborazione eseguita in componente vogliamo ritornare il valore di output al componente padre.

Componente Padre

```
<app-send parameter="Send Component"></app-send>  
<app-send (incrementCountEvent)="addItem($event)"></app-send>  
<p>{{counter}}</p>
```

Componente Figlio

```
@Output() incrementCountEvent = new EventEmitter<number>();  
  
increment(){  
  this.counter++;  
  this.incrementCountEvent.emit(this.counter)  
}
```


Alias per gli output

- Possiamo definire un alias per il nostro event emitter, in questo modo l'evento da riconoscere sarà identificato da quell'alias.

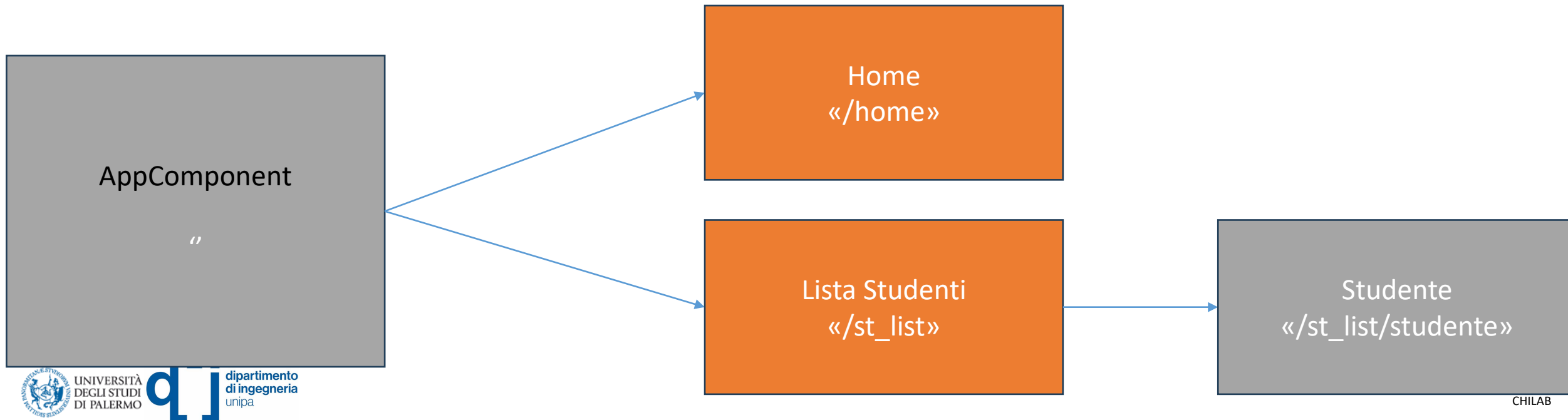
```
@Component({/*...*/})  
export class ChildComponent {  
  send = output({alias: 'sendMsg'});  
}
```

```
<app-child (sendMsg)=...>
```

Nel componente padre
intercettiamo l'evento in
questo modo usando l'alias

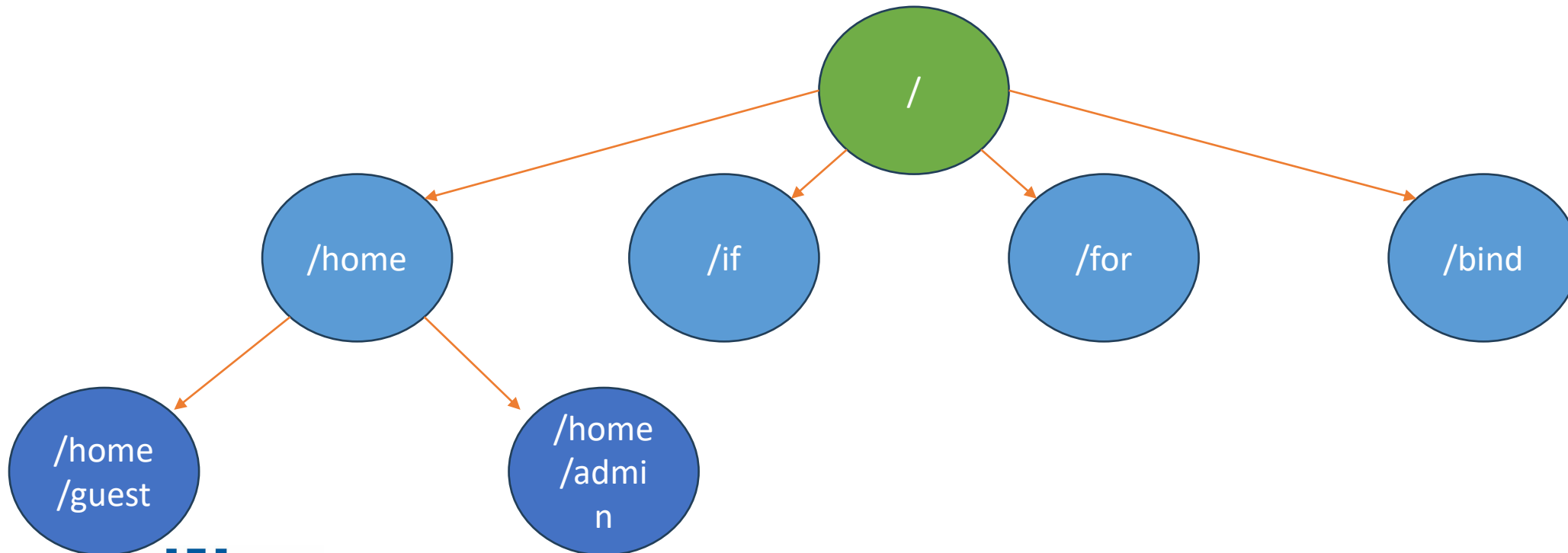
Routing

Nel momento in cui la nostra applicazione non è più una *single-page*, abbiamo la necessità di reindirizzare alle singole pagine che compongono la nostra architettura. Possiamo pensarli come dei path delle singole pagine, con «/» nodo di origine.



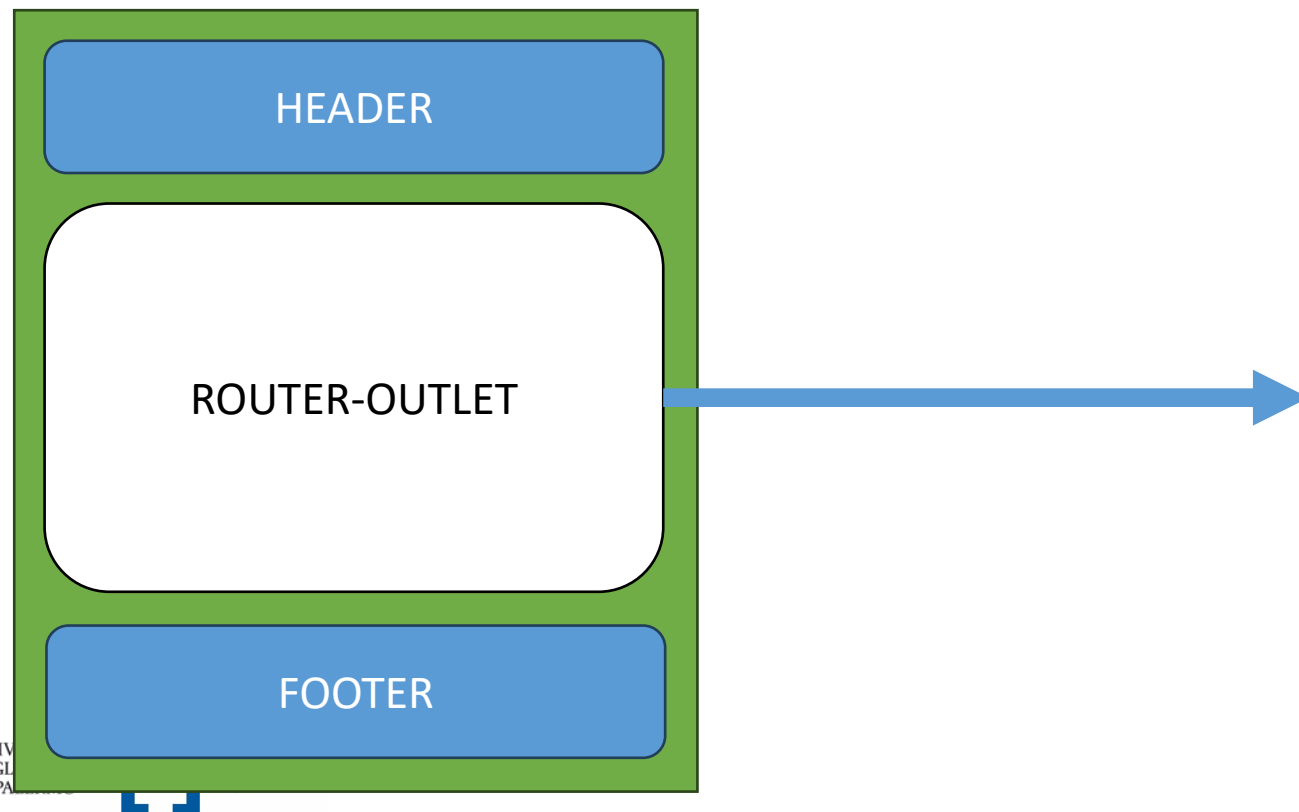
Routing

Probabilmente per avere una visione del funzionamento delle routes, possiamo anche pensarle come un albero.



Routing

Il routing associato alla struttura component-based delle applicazioni Angular, permette di sfruttare e riutilizzare il codice.



Strutturando un'applicazione in questo modo, nella zona dove viene indicato il ROUTER-OUTLET verranno caricate le pagine seguendo la logica definita dal routing. Se per esempio clicco su Home verrà inserito la grafica di quel componente, se clicco su HomeComponent e così via... mantenendo le altre parti della pagina invariate.

Router-Outlet

L'utilizzo del routerLink associato al router-outlet permette di caricare, nella sezione dove è definito quest'ultimo, soltanto il codice della pagina associata alla routes selezionata.

In questo la pagina non verrà caricata interamente ma soltanto una porzione di essa, rendendo l'esperienza più fluida e piacevole per l'utente.

```
<nav>
  <a routerLink="/home">Home</a>
  <br/>
  <a routerLink="/binding">Binding</a>
  <br/>
  <a routerLink="/add">Add</a>
  <br/>
  <a routerLink="/add-v2">Addv2</a>
  <br/>
  <a routerLink="/login">Login</a>
  <br/>
</nav>

<div class="container">
  <router-outlet></router-outlet>
</div>
```

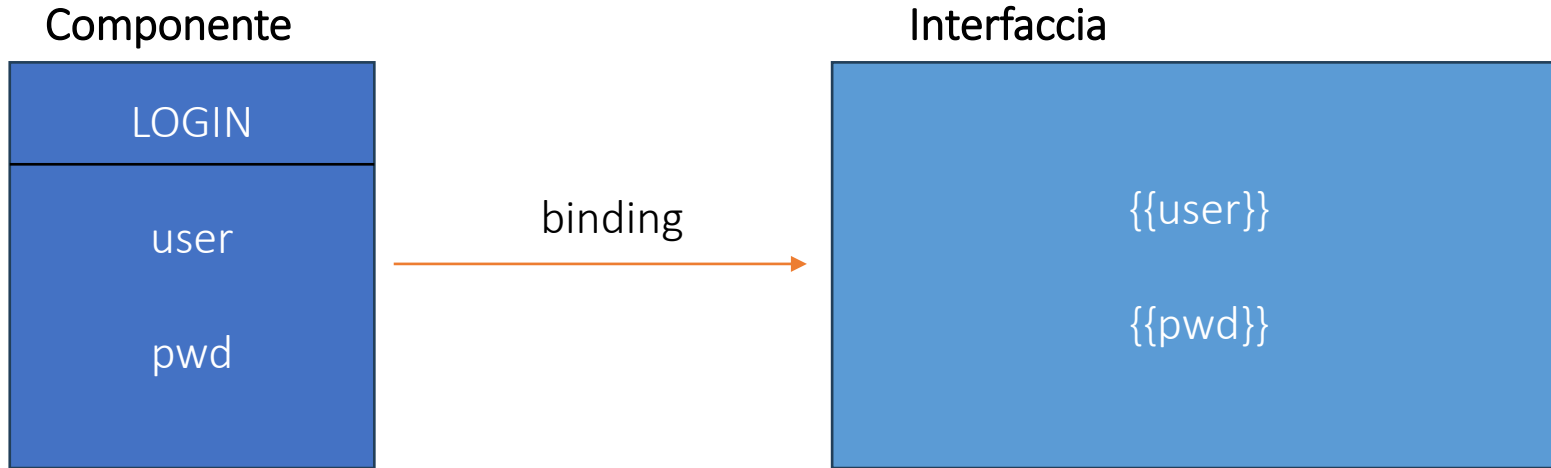
RouterLink

La gestione del redirect verso altre pagine è cruciale per le performance e l'esperienza utente. L'approccio tradizionale basato sugli **href** HTML provoca un refresh completo della pagina, che, in applicazioni complesse, può rallentare il sistema e consumare risorse in modo inefficiente. Per superare questo limite, Angular introduce **RouterLink**, un meccanismo che consente la navigazione tra le pagine senza ricaricare l'intera applicazione, migliorando così sia la velocità che l'efficienza complessiva.

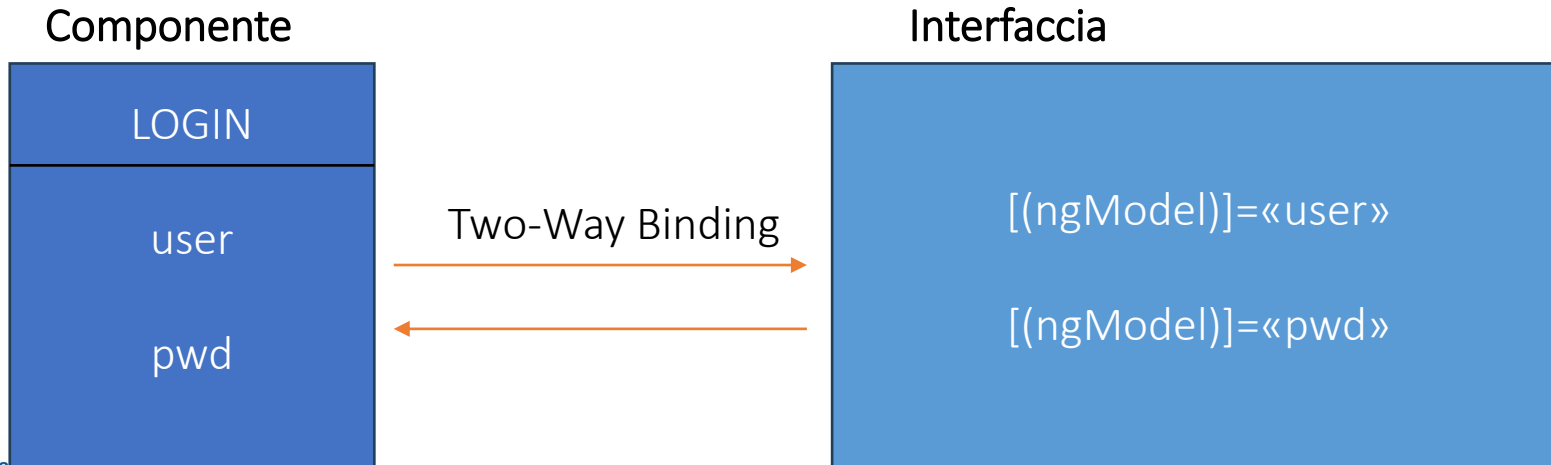
```
<p><a routerLink="/template" routerLinkActive="active-link" >templatePage</a></p>
```

Two-Way Binding

Binding permette
l'accesso alla
variabile



Binding permette
l'accesso alla
variabile e la
modifica della
stessa dalla GUI.



Form

I **form** sono uno degli strumenti fondamentali per creare interfacce utente interattive e dinamiche. In Angular, i form giocano un ruolo centrale nella raccolta, validazione e gestione dei dati forniti dagli utenti, rendendoli essenziali per applicazioni che richiedono input, come autenticazioni, registrazioni, o processi di checkout.

- **Raccolta dati:** Consentono agli utenti di inserire informazioni, come nomi, email, o preferenze.
- **Interattività:** Migliorano l'esperienza utente attraverso feedback immediati, come messaggi di errore in caso di input non valido.
- **Validazione:** Verificano che i dati forniti siano completi e corretti prima di inviarli al server.
- **Elaborazione dati:** Permettono di manipolare o trasformare i dati raccolti prima di utilizzarli.

Form- Template Driven

Questa definizione, però, non fa nulla di speciale se non definire la struttura della nostra form. Ciò che vorremmo ottenere sarebbe la possibilità di mappare il modello dei dati del componente, cioè la proprietà model, sulla form e viceversa, cioè **mappare il contenuto della form sul modello dei dati**. Per ottenere questo possiamo ricorrere al *two-way binding* chiamando in causa la direttiva ngModel

```
<form (ngSubmit)="onSubmit()" #loginForm="ngForm">
  <div>
    <label for="email">Email</label>
    <input type="email" id="email" name="email" required [(ngModel)]="email" />
  </div>

  <div>
    <label for="name">name</label>
    <input type="name" id="name" name="name" required [(ngModel)]="name" />
  </div>
  <br />
  <br />
  @if(loginFlag){
    <button type="submit" [disabled]="!loginForm.form.valid">Login</button>
  }
  @else{
    <button type="submit" [disabled]="!loginForm.form.valid">Registrati</button>
  }
</form>
```

Form- Template Driven

Questa immagine mostra degli input singoli, che vengono utilizzati per poter scrivere direttamente il valore delle variabili.

Questo meccanismo è possibile utilizzando il cosiddetto *Two Ways Binding*, che permette un mapping diretta fra la logica e la parte grafica della nostra applicazione.

```
<p>
  <label for="name">
    Nome Utente:
    <input id="name" type="text" [(ngModel)]="user" />
  </label>
</p>

<p>
  <label for="pwd">
    Password Utente:
    <input id="pwd" type="password" [(ngModel)]="pwd" />
  </label>
</p>
```

Form- Template Driven

In questo modo però non stiamo utilizzando una struttura unica che raggruppa insieme tutti i campi del nostro form. Quindi per riuscire a imporre eventuali controlli su:

- lunghezza min/max.
- Pattern di input

```
<p>  
  <label for="name">  
    Nome Utente:  
    <input id="name" type="text" [(ngModel)]="user" />  
  </label>  
</p>  
  
<p>  
  <label for="pwd">  
    Password Utente:  
    <input id="pwd" type="password" [(ngModel)]="pwd" />  
  </label>  
</p>
```

Form – Reactive Forms

Questo approccio crea un legame più profondo fra la GUI e il business model e andando a definire controlli specifici su valori inseriti.

```
<form [formGroup]="myForm">
  <p>
    <label for="name">
      Nome Utente:
      <input id="name" type="text" formControlName="txtName" />
    </label>
  </p>

  <p>
    <label for="pwd">
      Password Utente:
      <input id="pwd" type="password" formControlName="txtasseord" />
    </label>
  </p>

  <button type="button" [disabled]="!myForm.valid">Invia</button>
</form>
```

```
import { FormControl, FormGroup, ReactiveFormsModule } from '@angular/forms';

@Component({
  selector: 'app-reactive',
  standalone: true,
  imports: [ReactiveFormsModule],
  templateUrl: './reactive.component.html',
  styleUrls: ['./reactive.component.scss']
})
export class ReactiveComponent {

  myForm: FormGroup;
  constructor() {
    this.myForm = new FormGroup({
      txtName: new FormControl(),
      txtPassword: new FormControl(),
    });
  }
}
```

Gestione degli stati del form

Angular permette la gestione di eventi o stati del form tramite il CSS. Ecco una lista di alcuni stati gestiti:

- .ng-valid
- .ng-invalid
- .ng-pending
- .ng-pristine
- .ng-dirty
- .ng-untouched
- .ng-touched
- .ng-submitted

Così facendo possiamo per esempio disabilitare il submit finché i campi del form non sono stati tutti validati. Questo approccio permette di effettuare un primo check sui valori che saranno utilizzati in altre parti del front-end o del back-end.

Pipes

Le **pipes** in Angular sono strumenti che consentono di trasformare e formattare i dati direttamente nei template, senza bisogno di manipolarli nel codice del componente.

Esistono pipes di default come quelle mostrate nell'esempio:

- {{'hello' | lowercase }} → OUTPUT: hello
- {{'hello' | uppercase }} → OUTPUT: HELLO

Possiamo anche concatenare più pipes insieme :

<p>{{ 1234.56 | currency:'USD' | uppercase }}</p> → OUTPUT:"\$1,234.56"

Pipes

Nel caso di necessità specifiche possiamo definire delle pipe customizzate

```
import { Pipe, PipeTransform } from '@angular/core';  
@Pipe({ name: 'capitalize' })  
export class CapitalizePipe implements PipeTransform {  
  transform(value: string): string {  
    return value.charAt(0).toUpperCase() + value.slice(1).toLowerCase();  
  }  
}
```

<p>{{ 'angular' | capitalize }}</p> → 'Angular'

Services

In Angular possiamo definire delle particolari classi, definiti Services. Questi, vengono utilizzati perché riescono a creare delle zone condivise nel quale inserire variabili, funzioni, ecc... al quale molti componenti possono o devono accedere.

L'utilizzo del services risulta molto utile quando:

- Utilizziamo delle API, quindi molti componenti devono effettuare operazioni di CRUD.
- Dobbiamo condividere dei dati fra diversi componenti, che non stanno in una relazione padre-figlio.

Iniettare il Services dentro un Componente

Esempio di Services, dove vengono definite le chiamate API al server.

Il service verrà iniettato nei singoli componenti andando a definirlo nel costruttore.

Es.

```
constructor(private loginService:  
LoginService) { }
```

All'interno della classe si potrà accedere ai metodi tramite il
`this.loginService.metodo()`

```
@Injectable({  
  providedIn: 'root',  
})  
  
export class LoginService {  
  
  private baseUrl = 'http://localhost:3000'; // Modifica con l'URL del tuo backend  
  
  constructor(private http: HttpClient) { }  
  
  login(credentials: { username: string, email: string }): Observable<any> {  
    return this.http.post(`${this.baseUrl}/login`, credentials);  
  }  
  
  register(credentials: { username: string, email: string }): Observable<any> {  
    return this.http.post(`${this.baseUrl}/register`, credentials);  
  }  
}
```

Usare il Service come «zona» condivisa

- Un altro uso del Service è la possibilità di utilizzare delle variabili in condivisione fra i componenti in cui viene iniettato.

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

  data: {id: number,
    name: string,
    prezzo: number,
    img_path: string}[] = [];

  constructor() { }

  getData(){
    return this.data;
  }

  setData(data: {id: number,
    name: string,
    prezzo: number,
    img_path: string}[]){
    this.data = data;
  }

  setItem(p: {id: number,
    name: string,
    prezzo: number,
    img_path: string}){
    this.data.push(p);
  }
}
```