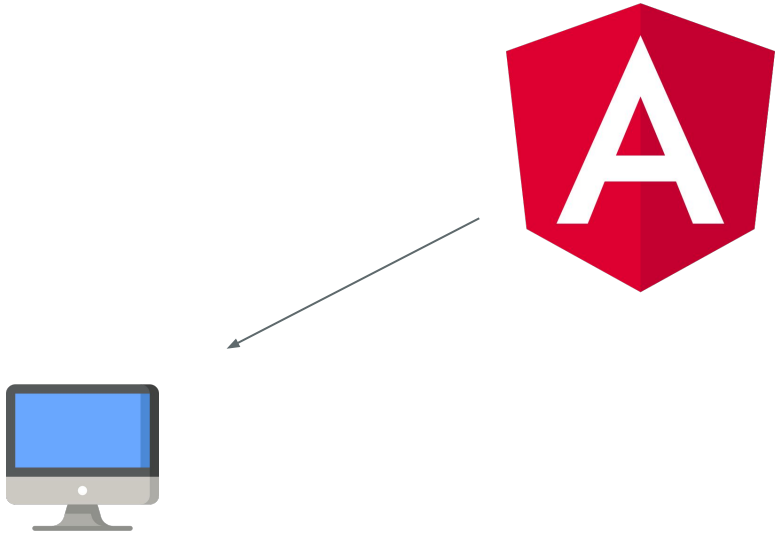

ANGULAR

Cos’è Angular



Framework Javascript

Costruzione interfacce e pagine web interattive

Raccolta di Tool

CLI, debugging, Plugins

Set Up Angular Development Enviroment

Node.Js : https://nodejs.org (Latest Version)

Npm: (incluso in Node.js) è un package manager che scarica e installa le librerie necessarie all'applicazione

Visual Studio Code (IDE): https://code.visualstudio.com/

- **Angular Language Service** (fornisce suggerimenti durante la stesura del codice, import automatici)
- **Angular Essential by John Papa** (bundle di tool)

Angular CLI: lanciare da terminale il seguente comando

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

ARCHITETTURA

HTML + TYPESCRIPT

Gli elementi più importanti sono:

- *NgModules*. Un'applicazione Angular è definita da set di NgModules. Tutte le applicazioni hanno come punto di partenza il root module
- *Components*. Definiscono le *"views"* e utilizzano i "services"

Views: parti di html che Angular può prendere e modificare seguendo una logica

Services: possono essere iniettati nei components come dipendenze. Rappresentano le funzionalità delle views. Grazie ai services il codice diventa modulare, riutilizzabile.

Creazione Nuova APP (Modern Angular - No Legacy)

Lanciare il seguente comando per creare una **nuova App**:

```
ng new mia-app
```

Con questo comando verranno scaricati i pacchetti necessari al funzionamento dell'app, insieme ai node-modules della cartella app.

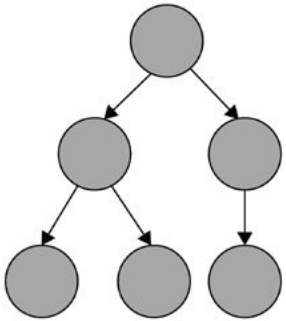
Lanciare l'App (da terminale) :

```
ng serve --open
```

COMPONENT

I component sono alla base di un'app Angular. Controllano le parti della nostra pagina chiamate **views**.

Un'applicazione consiste in un'alberatura di component che possono interagire e comunicare tra di loro



Il comando utilizzato per la creazione dei component:

ng generate <type> <name>

Es:

ng generate component nomeComponent

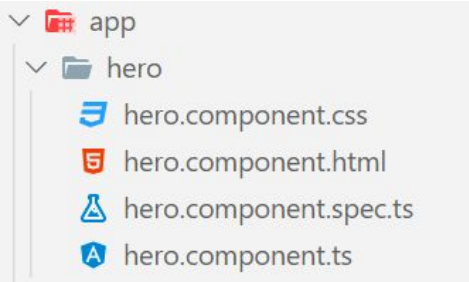
Per quanto riguarda i **type** disponibili: https://angular.io/cli/generate#nggenerate

Creazione Nuovo Component

Creazione **nuovo Component**:

```
ng generate component nomeComponent
```

Verrà creata la seguente alberatura all'interno della cartella app:



In automatico verrà inserita la parola .component nei file
.css - **Style**
.ts - **Logica del component**
.html - **Template**
.spec.ts - **Test Unit**

Versione abbreviata:

```
ng g c nomeComponent
```

Configurare Component

Un component è tipicamente una classe Ts marcata con il **decorator @Component** importato da @angular/core

selector: è il nome del component per poterlo identificare all'interno dello HTML Template
templateUrl: path relativo del file di template
styleUrls: path relativo dei fogli di stile .css

```
@Component({
  selector: 'app-hero',
  standalone: true,
  imports: [],
  templateUrl: './hero.component.html',
  styleUrl: './hero.component.css'
})
```

A questo punto è possibile visualizzare il component all'interno dell'html desiderato, attraverso l'utilizzo del suo selettore.

```
<app-hero></app-hero>
```

ATT: ogni component può essere "renderizzato" n volte e all'interno di altrettanti components differenti.

Considerazioni sulla proprietà **`standalone: true`**

Questa caratteristica è stata introdotta in Angular 16, per cui attualmente è facile trovare progetti sviluppati prima di questa versione. In questi progetti non é presente questa caratteristica, o meglio, è come se fosse settata su false

Built-in Directives

Directives: aggiungono funzionalità, permettono di manipolare il DOM, modificano il comportamento degli elementi HTML .

ATT: i component per loro natura sono considerati directives ma con Template

- **ngIf (Legacy) - @if :** aggiunge o rimuove porzioni di HTML dall'albero del DOM
- **ngFor(Legacy) - @for** : iteratore di oggetti o liste legato al template
- **ngSwitch** : crea uno switch tra i possibili template

- **ngStyle**: fornisce regole css a elementi HTML
- **ngClass**: imposta classi css dinamicamente
- **ngModel**

ngModel

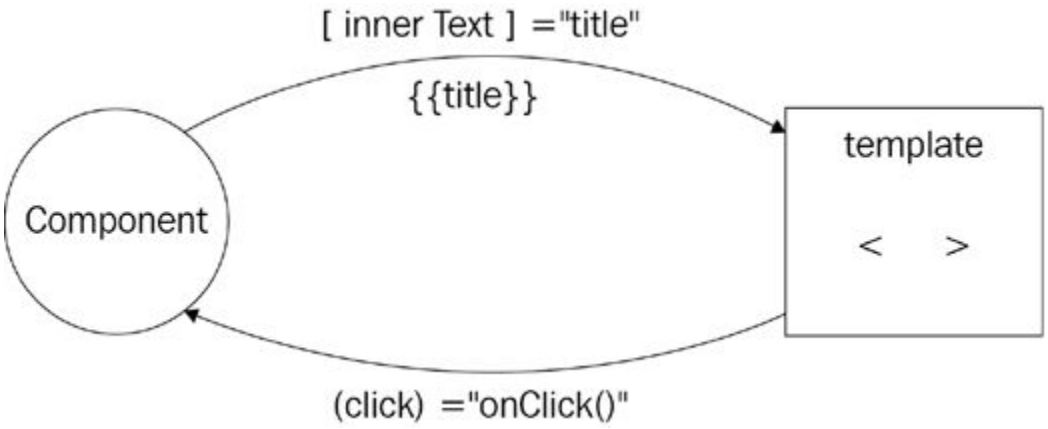
ngModel è una direttiva che lega il model ad un form, è particolare poiché simula il **two-way data binding**, tipico di Angular1 (JS).

Si utilizza la directive **ngModel** per tenere in sincro la variabile d'istanza con la view.

Property Binding & Event Binding

Property Binding: utile per mostrare proprietà della classe nel template, è possibile eseguirlo tramite **{{ }}** oppure tramite le parentesi **[ ]** utilizzate negli attributi del template

Event Binding: permette di recuperare dal template dati e passarli al component. Attuabile tramite l'evento **(click)** usato come attributo nel template



Comunicare con altri Component

I Component di Angular espongono una API pubblica che gli permette di comunicare con altri component attraverso delle proprietà di **Input** usate per iniettare dati. Vengono utilizzate delle proprietà di **Output** utilizzate insieme agli **event listener**.

@Input (Parent ↠ Child)

Il decorator **@Input( )** (seguito dal nome della proprietà) è utile per passare dati da un component ad un altro "più giù" nella gerarchia. (**Parent -> Child) (app.component -> child.component)**

1. Si definisce con il decorator @Input all'interno della classe del component
2. Importare da **@angular/core** il pacchetto Input
3. Iniettare attraverso l'uso della Property Binding dal template dell'AppComponent utilizzando il selettore del nuovo Component

   es:

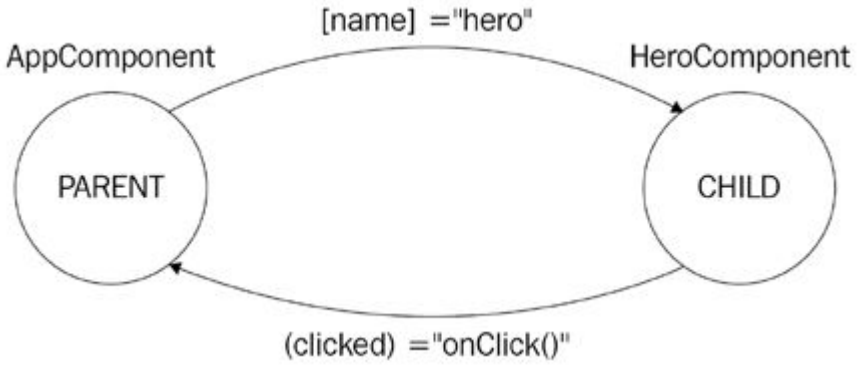
   <app-hero [nome]="Batman"></app-hero>

@Output (Child ⇢ Parent)

Utile in situazioni inverse rispetto a @Input, ovvero un trasferimento di eventi da child a parent **(Child -> Parent) (Child.component -> App.component).**

Utilizzo il decorator **@Output** all'interno della classe del component Child ricordandomi di importare il pacchetto Output from @angular/core.

Es: utilizzo di @Output attraverso un **EventEmitter()** che intercetta un click nel component child e passa l'informazione al component Parent il quale implementerà un metodo per mostrare il dato.



Component Lifecycle

I “**Lifecycle Hook**” permettono di controllare le fasi di vita di un component e applicare una logica personalizzata per ogni situazione. Ogni hook presuppone l’utilizzo di un’interfaccia che definisce i metodi che verranno implementati in futuro (l’utilizzo delle interfacce non è obbligatorio ma è una buona pratica).

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

Tutti disponibili in @angular/core

Es:
OnInit viene aggiunto in automatico dalla classe e porta con sé l’implementazione del metodo **ngOnInit**.
Stessa cosa succede quando utilizziamo altri hook

OnInit è utile quando inizializziamo un component utilizzando dati che arrivano da una fonte esterna.

OnDestroy utile quando si rimuove un component dall’albero del DOM

OnChanges utilizzato quando cambia il valore di un determinato binding

Manipolare dati con |

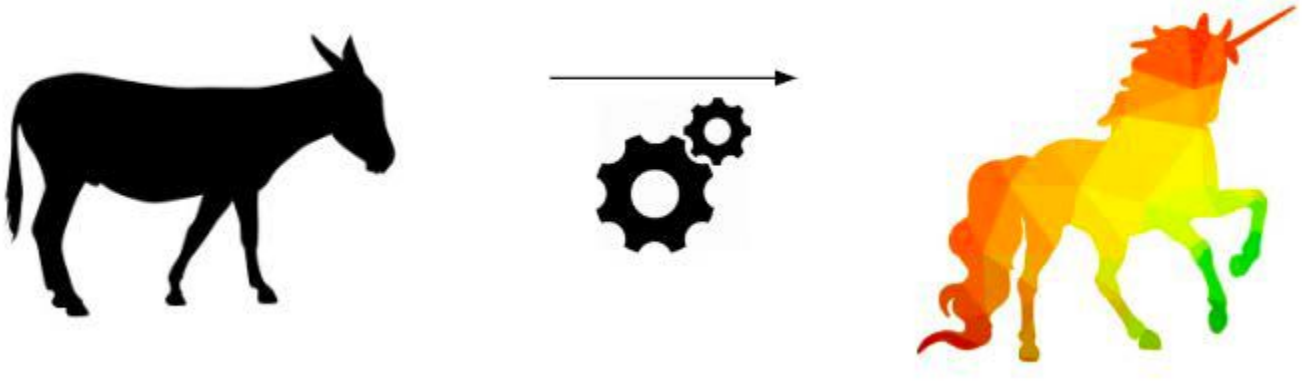
Pipes: permettono di utilizzare e trasformare le informazioni del template. La sintassi è molto semplice, si utilizza il simbolo | con un'espressione a seguire

| uppercase **|lowercase**

| slice:1:3 **|date**

|currency: EUR **|currency: USD**

| json



Custom Directives

Pipes e Directives possono essere personalizzate e create a piacimento.

Creare una directive personalizzata per ottenere il property binding e mostrare dati dinamici.

ng generate directive <nomeDirective>

La directive creata viene creata nella cartella principale dell'app e viene inserita nel file AppModule

Angular Forms

Pensati per modificare i dati sia sulla pagina sia sul server, i form sono un aspetto fondamentale in Angular. Abbiamo a disposizione alcuni strumenti per il controllo, la validazione e il testing dei form.

- **FormControls & FormGroups**: incapsulano gli input dei nostri form restituendo un oggetto con il quale lavorare
- **Validators**
- **Observers** : permettono di lavorare ai form guardando i cambiamenti che avvengono.

Oss: per utilizzare i form bisogna importare nell'app.module.ts le libraries relative : FormsModule, ReactiveFormsModule. Queste ci permetteranno di utilizzare le directives

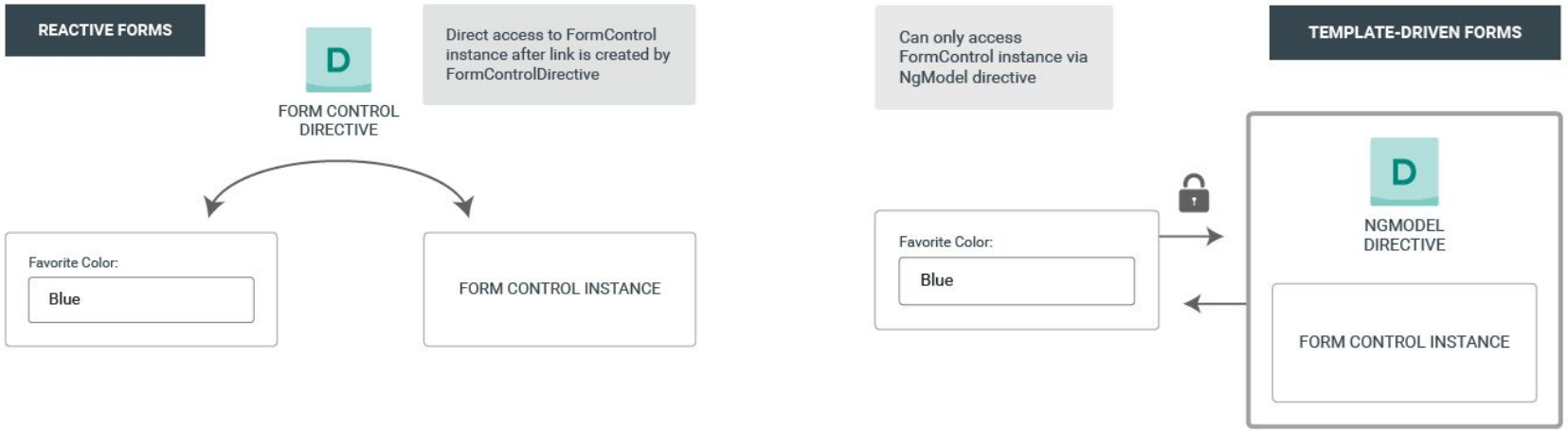
- *ngModel* e *ngForm*
- *formControl* e *ngFormGroup*

e molte altre

Reactive vs. template-driven Forms

Angular permette di gestire i Form in due modi:

https://angular.io/guide/forms-overview



Dependency Injection

La **Dependency Injection** fa parte dei Design Patterns, condivisa con altri linguaggi.

Al crescere di un'applicazione, le nostre entities richiederanno sempre di più istanze di altri oggetti, meglio conosciuti come **dependencies**.

L'azione di passare queste dependencies verso un' entità "consumer" è chiamato **injection**. Il "consumer" non sa nulla riguarda come istanziare le **dependencies**, è solo a conoscenza dell'interfaccia che viene implementata per utilizzarle.

Questo meccanismo è utile per maneggiare dati non statici, cosa che nelle app in Angular accade spesso dato che solitamente i dati provengono da API esterne o altri **services**. (non come l'esempio della lista di heroes schiantata così com'è)

Nota: i component in Angular devono essere utilizzati solo per la logica della presentazione, non devono occuparsi di recuperare dati. Questo compito viene delegato ai **services** i quali attraverso la **Dependecy Injection** passano questi dati al component

Delegare con i Services

ng generate service nome

Questo comando crea un file **nome.service.ts**

I **Services** sono una classe di Angular avente il decorator **@Injectable** il quale identifica la possibilità di essere iniettato in un altro component o in un altro service a sua volta.

All'interno del service sviluppiamo la logica che può essere utilizzata dal nostro component.

Passaggi fondamentali:

- creare la logica nel service
- iniettare il service con modificatore **private** all'interno del costruttore del component che lo utilizzerà
- sfruttare i metodi del service per ottenere i dati