
Angular

Angular

1. Introduzione

ngular – Vantaggi rispetto AngularJS

Angular2 introduce una lunga serie di **“breaking changes”** che lo rendono un framework molto diverso dal predecessore, AngularJS.

- Controllers e $scope sono stati sostituiti da **components** (componenti) e **directives** (direttive)
- Modularitá del software grazie alla produzione di **modules** (moduli)
- Sintassi piú semplice per il binding di proprietá ed eventi
- Reactive programming con RxJS
- TypeScript come linguaggio di sviluppo

ngular – Typescript

Un superset di JavaScript ECMAScript 2015 (ES6) retrocompatibile con ES5 (JavaScript). Include quindi tutti I maggiori benefici di ES6, tra I quali:

- Lambda Functions
- Iterators
- For…of
- …etc

Tra le features più interessanti introdotte da TypeScript ci sono la **tipizzazione di variabili e proprietà** nonchè l'introduzione degli **access modifiers.**

ngular – Requisiti software

- Node.js >= 4.x.x
- npm >= 3.x.x
- typescript

In ambiente Microsoft Windows é consigliato utilizzare un gestore di pacchetti come **Chocolatey**. Grazie a questo package manager possiamo installare le dipendenze software da riga di comando digitando:

choco install nodejs

In alternativa é possibile scaricare l'installer da nodejs.org

Per installare TypeScript digitare:

Npm install –g typescript

ngular – Inizializzazione

É possibile inizializzare un'app Angular2 in due modi:

- Copiando un repository github contenente lo skeleton di una app e le dipendenze npm giá configurate
- Attraverso l'uso del componente @angular/cli

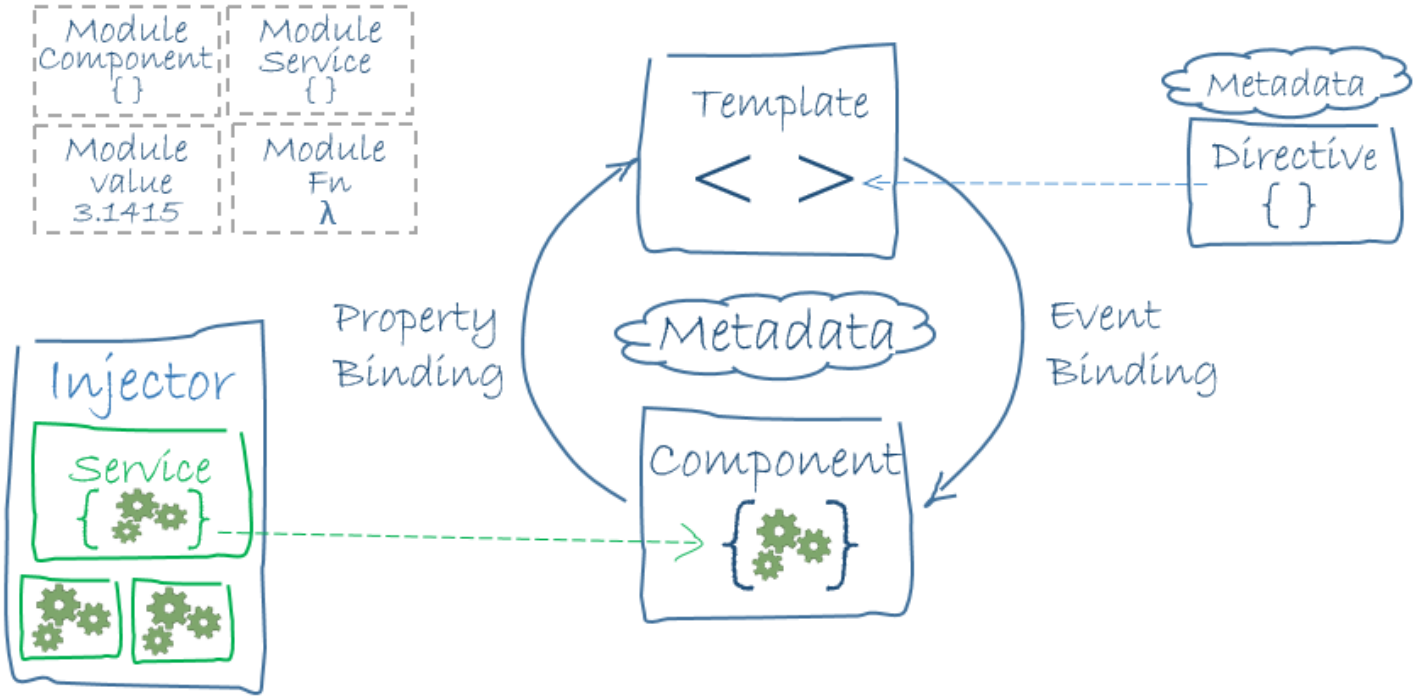
I vantaggi principali nell'uso della CLI (Command Line Interface) sono:

- Utilizzo di un comando da terminale senza dover ricordare l'indirizzo del repository da clonare
- Impiego immediato di uno strumento utile anche per la creazione on-the-fly di moduli, componenti, provider…

ngular

2. Architettura di un’applicazione

ngular – Struttura dell'applicativo



ngular – Moduli

Rendono l'app **modulare**

- I moduli, chiamati anche ngModules, sono un insieme di componenti, servizi e direttive che uniti insieme compondono una **feature** dell'applicativo

Esiste almeno un modulo

- Le applicazioni Angular2 sono composte da **almeno un modulo**, chiamato **root module**. Applicazioni particolarmente

Sono classi con decorator @NgModule

- Un modulo non é altro che una classe decorata con I metadata presenti nel decorator @NgModule
- Un modulo Angular é differente da un modulo JavaScript!

ngular – Moduli

@NgModule é un decoratore che accetta un set di metadati e che descrive il modulo:

- Declarations: l'insieme di classi legate alle view del modulo (componenti, direttive e pipes)
- Exports: gli oggetti ai quali possono accedere altri moduli – **il root module non necessita di questa keyword**
- Imports: i moduli dai quali si vuole attingere a delle classi esportate
- Providers: i servizi utilizzati nel modulo
- Bootstrap: il componente iniziale dell'applicazione – **deve essere dichiarato solo nel root module**

ngular – Componenti

Gestiscono una view

- Il componente é delegato alla gestione dell'interazione tra l'utente ed una porzione dello schermo che prende il nome di **view**

Sono classi con decorator @Component

- Un componente consiste in una classe decorate con I metadata presenti nel decorator @Component

Intervengono nel lifecycle dell'applicativo

- Angular crea, aggiorna e distrugge I componenti dipendentemente all'interazione utente
- I componenti possono eseguire delle operazioni nelle diverse fasi del lifecycle dell'applicazione

ngular – Templates

Sono la parte visibile dei componenti

- I template sono il complemento grafico dei componenti
- Il comportamento di ciascun template é determinato dalle regole definite nel componente associato

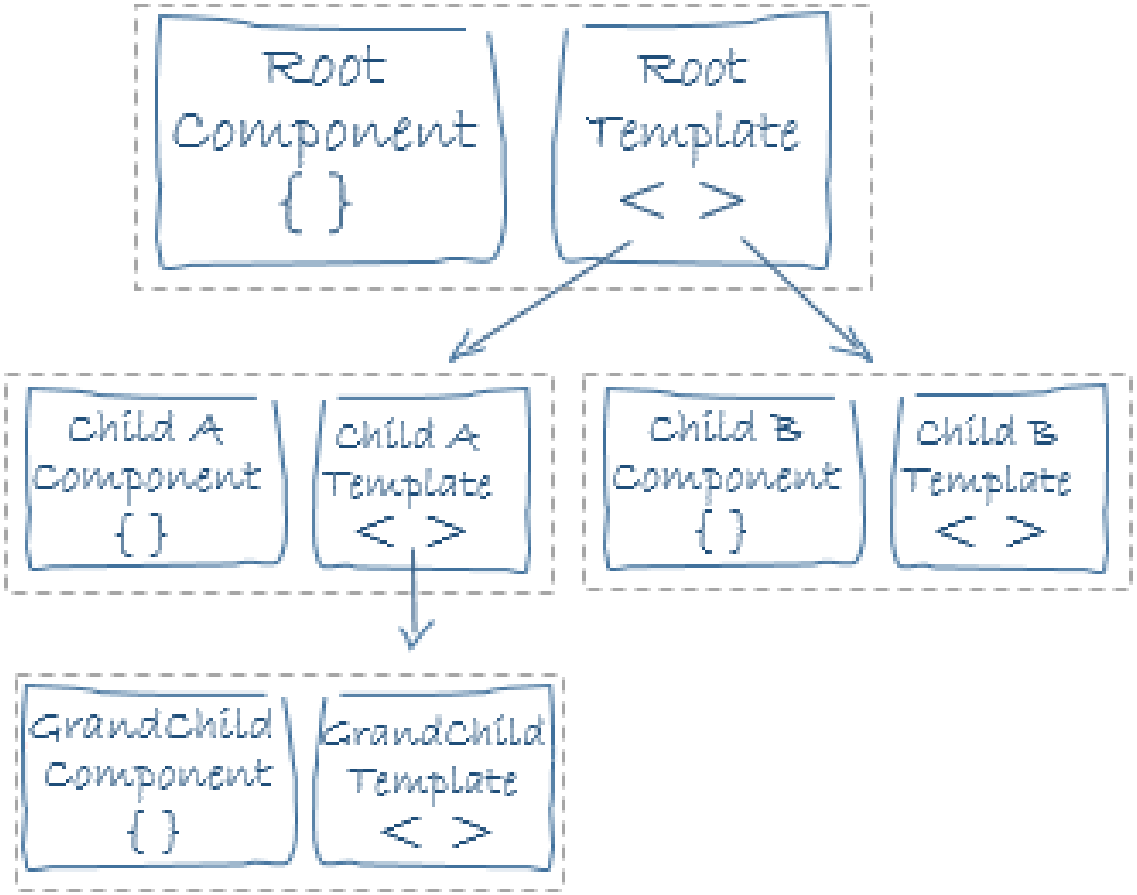
Un HTML piú… ricco!

- Utilizzano I tipici tag HTML
- Il codice HTML é arricchito dalle istruzioni di Angular (controllo, iterazione, binding di eventi e proprietá)

Possono inglobare altri componenti

- É possibile inglobare all'interno della view altri componenti inserendo il tag ad essi associati

ngular – Templates



ngular – Metadata

I metadata servono a definire ad Angular come elaborare una classe. Nel codice possono essere individuati:

- **Dalla posizione**: vengono dichiarati sempre **subito prima** della dichiarazione di una classe o di una proprietá
- **Dalla simbologia**: vengono richiamati attraverso il simbolo @

Accettano in input un oggetto che valorizza le proprietá del metadata che si vuole utilizzare.

I template Angular sono **dinamici**

- Le directives sono delle classi che trasformano gli elementi del DOM secondo le istruzioni contenute al loro interno
- Possono aggiungere / rimuovere elementi del DOM oppure modificarne gli attributi

Sono classi con decorator @Directive

- Esattamente come per Moduli e Componenti, le direttive sono classi JavaScript decorate con un apposito set di metadati

ngular – Servizi

A cosa servono?

- Sono classi con uno scopo ben definito
- Offrono una serie di metodi per raggiungere uno specific scopo

Sono “iniettabili” nella DI di Angular

- Il decorator @Injectable comunica al framework la possibilitá di iniettare il servizio nella DI di Angular
- Angular non ha una classe “base” per I servizi: il decorator, in questo caso, ha il solo scopo di qualificare la classe come “iniettabile”

Alcuni esempi: un servizio puó…

- Svolgere tutte le funzionalitá di logging incluse in un’applicazione
- Essere delegato all’interrogazione asincrona di un’interfaccia ReST
- Elaborare dei dati, eseguendo calcoli sulla base di specifici algoritmi

I servizi hanno largo impiego in un'applicazione. I Componenti, infatti, non dovrebbero contenere (ad esempio) logiche di validazione, procedure di logging ed implementazioni di chiamate ad interfacce: questo tipo di attivitá **devono** essere lasciate ai **servizi**.

RICORDA

I COMPONENTI SI LIMITANO A FARE DA TRAMITE TRA IL TEMPLATE E LA LOGICA APPLICATIVA (SERVIZI)

ngular – Root module

In Angular, un **modulo** definisce l'interazione tra le differenti parti dell'applicativo. Qualsiasi applicazione sviluppata con Angular2 ha **almeno un modulo**: il **root module** che viene richiamato all'atto della procedura di **bootstrap** (inizializzazione).

src/app/app.module.ts

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

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

ngular – Root module

Il decorator @NgModule

- Identifica l'AppModule come una classe Angular di tipo Module (chiamata anche NgModule)
- Essendo un decorator, prende in Input un oggetto di "metadata"

I metadata

- **imports:** importiamo il BrowserModule, parte integrante di Angular, che permette alle applicazioni di essere eseguite all'interno del Browser
- **declarations:** dichiariamo come parte di questo modulo il componente AppComponent
- **bootstrap:** definiamo come **entry point** dell'applicativo il componente AppComponent

ngular – main.ts

Contiene la procedura di **bootstrap** dell'applicazione e ne gestisce la compilazione (di default con sistema Just In Time). Serve a registrare i **selector** dei componenti presenti nell'applicazione all'interno del DOM sottoforma di TAG.

src/main.ts

```
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
import { AppModule }              from './app/app.module';

platformBrowserDynamic().bootstrapModule(AppModule);
```

Angular

3. Componenti

ngular – Definizione di un Componente

I componenti sono, insieme ai **template**, parte fondamentale per la creazione di una View. Vengono definiti attraverso il decorator **@Component** e sono richiamati nei template attraverso un TAG HTML che li identifica all'interno del DOM.

src/app/app.component.ts

```
1.  import { Component } from '@angular/core';
2.
3.  @Component({
4.    selector: 'my-app',
5.    template: `
6.      <h1>{{title}}</h1>
7.      `
8.  })
9.  export class AppComponent {
10.   title = 'A Title';
11. }
```

ngular – @Component

Il decorator @Component

- Identifica l'AppComponent come una classe Angular di tipo Component
- Essendo un decorator, prende in Input un oggetto di "metadata"

I metadata

- **selector:** identifica il Componente all'interno del DOM mediante l'associazione di un tag
- **template / templateUrl:** permette di definire il template indicandone il codice (template) o l'indirizzo ad un file html (templateUrl)
- **styles / stylesUrls:** permette di definire un array di **style css** (styles) o un array di indirizzi a file css (styleUrls)
- **directives:** consente di dichiarare un array di Componenti e Direttive utilizzati nel Template

ngular – Componenti con Angular CLI

Angular CLI permette la creazione rapida di Componenti Angular

Un Componente Angular2 si inizializza con il comando:

ng generate component [nome]*

* Sostituire [nome] con il nome del proprio componente in **dash case**

I parametri supportati dal comando sono disponibili sulla wiki di AngularCLI

ngular – Vantaggi della CLI

L’impiego della CLI di Angular permette di ridurre il tempo di esecuzione di procedure ripetitive grazie all’automatizzazione di alcuni processi. Grazie ai parametri disponibili con il comando **generate** é possibile (tra l’altro):

- Definire il modulo di appartenenza di un componente
- Specificare se vengono utilizzati style e template inline
- Generare dei boilerplate di test funzionali
- E molto altro…

Grazie alla CLI é possibile creare non soltanto Componenti, ma qualsiasi tipo di struttura relativa ad un applicativo Angular2 (Direttive, Moduli, Servizi…)

Essendo i **componenti** delle **semplici classi JavaScript**, possiamo definire delle **proprietá** e dei **metodi** che concorreranno a gestire rispettivamente lo **stato** del componente ed il relativo **cambiamento di stato**.

TypeScript introduce l'**access control**, sia sulle proprietá che sui metodi, mediante l'uso delle ben note keyword *private*, *public* e *protected*. Inoltre è possibile **tipizzare** le proprietá ed i parametri nella firma dei metodi, nonché il risultato restituito dai metodi stessi.

ngular – Interpolazione

Le **proprietá** sono una parte fondamentale della View e possono essere mostrate nel template attraverso il sistema di interpolazione.

Con l'**interpolazione** si introduce il **primo tipo di data binding** disponibile in Angular: permette di mostrare la proprietá di un componente all'interno del template indicando il nome della proprietá all'interno di doppie parentesi graffe…

```
<h1>{{ title }}</h1>
```

ngular – Interpolazione

Quando si utilizza l'interpolazione viene applicato un controllo sullo stato della proprietá indicata. Questo significa che al mutare dello stato della proprietá, il template viene aggiornato con un nuovo valore.

src/app/app.component.ts

```
1.  import { Component } from '@angular/core';
2.
3.  @Component({
4.    selector: 'my-app',
5.    template: `
6.      <h1>{{title}}</h1>
7.      `
8.  })
9.  export class AppComponent {
10.   title = 'A Title';
11. }
```

Angular

4. Gestione dell’interazione utente

Le **azioni** eseguite dall'**utente** generano eventi all'interno del DOM. Generano eventi, ad esempio:

- Pressione di un bottone
- Click su elementi come i link
- Modifica di input field

È possibile restare in ascolto per uno specifico evento su un elemento utilizzando il **binding agli eventi**.

ngular – Event Binding

Al contrario dell'interpolazione…

- L'**event binding** gestisce un flusso dati da un elemento del template al componente
- Intercetta l'evento specificato ed esegue il metodo definito nella firma del binding

Come si definisce?

- É definito su un elemento del template
- La sintassi si compone in (evento)="metodo()"
- Una sintassi alternativa nota come **canonical form** prevede l'utilizzo del prefisso on-evento="metodo()"

ngular – Event Binding

È possibile **prelevare informazioni** riguardo l'evento generato dall'utente: il DOM infatti genera un **payload ricco di informazioni** ogni qual volta accade qualcosa all'interno della View. Questo oggetto è intercettabile passando un particolare parametro al metodo in ascolto:

$event

src/app/keyup.components.ts (template v.1)

```
template: `
  <input (keyup)="onKey($event)">
  <p>{{values}}</p>
`
```

ngular – Event Binding

Quando un utente genera l'evento specificato all'interno del template, Angular istanzia una variabile $event che contiene informazioni circa l'evento appena generato. È necessario aggiornare la firma del metodo in ascolto in modo opportuno…

src/app/keyup.components.ts (class v.1)

COPY CODE

```
export class KeyUpComponent_v1 {
  values = '';

  onKey(event: any) { // without type info
    this.values += event.target.value + ' | ';
  }
}
```

Non tutti i Payload sono uguali tra loro!

I Payload generati a seguito di un evento possono differire in parte da evento ad evento. Tipizzare un payload con «any» non permette di avere contezza delle proprietá dell'oggetto. Fortunatamente è possibile **specificare il tipo di payload atteso in base all'evento.**

src/app/keyup.components.ts (class v.1)

COPY CODE

```
export class KeyUpComponent_v1 {
  values = '';

  onKey(event: any) { // without type info
    this.values += event.target.value + ' | ';
  }
}
```

ngular – Generare eventi

La generazione di eventi personalizzati avviene mediante l'uso della classe **EventEmitter**. Quando una classe implementa in una sua proprietá un oggetto EventEmitter, puó **emettere un evento** attraverso il suo metodo **emit** e trasmettere un **payload di informazioni personalizzato.**

Quando si dichiara un EventEmitter bisogna specificare il tipo di payload che si vuole emettere nella propagazione dell'evento.

src/app/keyup.components.ts (class v.1)

COPY CODE

```
export class KeyUpComponent_v1 {
  values = '';

  onKey(event: any) { // without type info
    this.values += event.target.value + ' | ';
  }
}
```

ngular – Generare eventi

src/app/hero-detail.component.ts (deleteRequest)

```
// This component make a request but it can't actually delete a hero.
deleteRequest = new EventEmitter<Hero>();

delete() {
  this.deleteRequest.emit(this.hero);
}
```

COPY CODE

Il componente che gerarchicamente si puó identificare come **figlio** implementa una proprietá *public* di tipo **EventEmitter.** Il tipo di payload viene specificato tra i caratteri < e >.

new EventEmitter<tipo>();

ngular – Generare eventi: @Output()

Per permettere agli altri componenti di restare in ascolto su una proprietá, bisogna decorare questa come una **proprietá in Output.** Il decorator @Output permette appunto di definire il flow di una proprietá **dall'interno verso l'esterno.**

Gli EventEmitter, dovendo inviare dati all'esterno del componente, devono tassativamente essere decorati come proprietá in Output. Il decorator @Output si utilizza anteponendolo all'accessor della proprietá. Ad esempio:

```
@Output() deleteRequest: EventEmitter<Hero>;
```

ngular – Generare eventi

Il componente gerarchicamente identificabile come **padre** resta in ascolto dell'evento del componente precedente, utilizzando la sintassi:

<selector (nomeEventEmitter)="metodo()"></selector>

```
<hero-detail (deleteRequest)="deleteHero($event)" [hero]="currentHero"></hero-detail>
```

Spesso gli eventi sono utilizzati per modificare lo stato delle proprietá di altri componenti. Questo genere di azioni causa **cambiamenti in cascata** su tutti gli elementi del DOM che monitorano lo stato della proprietá che subisce variazioni!

5. Passare dati ad un Componente

Il decorator @Input gestisce il flow in modo opposto al decorator @Output: qualora infatti un componente padre debba impostare una proprietá presente in un componente figlio, questa dovrá essere decorata con il decorator @Input.

@Input gestisce il flow dei dati **dall'esterno verso l'interno**

Come per @Output, il decorator @Input viene collocato prima dell'accessor di una proprietá.

```
@Input() public property: string;
```

ngular – Property Binding

Per poter associare una proprietá da un componente padre ad un componente figlio si utilizza il **Property Binding.**

```
<child [childProperty]=“parentProperty"></child>
```

Il Property Binding è usato anche per associare proprietá a tag HTML. Tutte le proprietá dei tag HTML vengono interpretate da Angular allo stesso modo di **proprietá decorate con @Input().**

ngular – Property Binding

É un binding di tipo **One Way**

- Gestisce il flusso dati di una proprietá da un componente padre alla proprietá di un componente figlio
- Se la proprietá muta di stato nel componente figlio, questa **non muta** lo stato della proprietá del componente padre

Come si definisce?

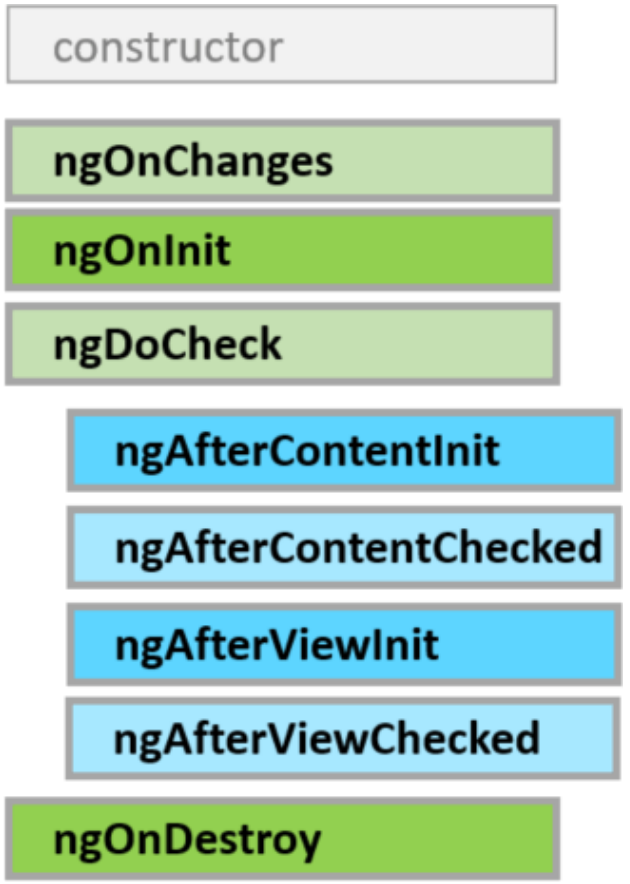
- Sulla proprietá di uno specifico elemento del template
- La sintassi si compone in <elemento [proprietáDelFiglio]="proprietáDelPadre">

Angular

6. Lifecycle

ngular – Lifecycle

Con il termine **lifecycle** si intende il **ciclo di vita** di **componenti** e **direttive**, ovvero di tutte quelle fasi che questi attraversano dall'atto della richiesta di una nuova istanza fino alla distruzione. I componenti, essendo delle direttive particolari, passano per un maggior numero di stati durante il loro ciclo vitale poiché devono renderizzare il contenuto all'interno del template.



Angular mette a disposizione delle **interfacce** che aiutano a rispettare l'interazione con questi eventi particolari. **Dichiarare l'implementazione dell'interfaccia non è obbligatorio ma è sempre fortemente consigliato!**

Le interfacce dei lifecycle

- Implementano ciascuna un singolo metodo chiamato esattamente come l'interfaccia, ma precedura dal suffisso **ng**.
- Fanno parte delle librerie **core** di Angular

Gli Hooks

- Sono tutti quei metodi che nascono per "agganciarsi" al ciclo di vita di component e direttive
- Vengono richiamati al termine dell'esecuzione del **costruttore**

ngular – ngOnChanges

È il **primo hook** chiamato al termine dell'esecuzione del **costruttore**. Viene eseguito ogni volta che le si verifica un **cambiamento** nelle **proprietá in input** e riceve un oggetto di tipo **SimpleChanges**.

Gli oggetto SimpleChanges sono particolarmente utili poiché contengono sia lo stato precedente della proprietá, sia quello attuale.

Gli oggetti passati in seguito agli eventi possono rivelarsi particolarmente utili: impara ad usarli al meglio analizzandoli nella console di debug del tuo browser preferito!

ngular – ngOnChanges

```
@Component({selector: 'my-cmp', template: `...`})
class MyComponent implements OnChanges {
  @Input()
  prop: number;

  ngOnChanges(changes: SimpleChanges) {
    // changes.prop contains the old and the new value...
  }
}
```

ngular – ngOnInit

È il **secondo hook** in ordine di chiamata. Viene chiamato in fase di controllo dei data binding. È l'hook ideale per eseguire chiamate a servizi esterni al fine di popolare la View di dati. ngOnInit determina, tra l'altro, l'avvenuta valorizzazione delle **proprietà in input** associate al componente.

Spesso viene confuso, per impiego, con il costruttore. Per comprendere bene l'uso del costruttore rispetto all'evento OnInit, basta associare il primo all'inizializzazione di proprietá ed all'**injection dei servizi.**

Viene richiamato una sola volta!

Questo hook puó essere chiamato in aggiunta ad ngOnChanges qualora non vengano rilevate modifiche allo stato dell'applicazione. All'interno di **ngDoCheck** è possibile **definire una strategia personalizzata di individuazione del cambiamento di stato di una o piú proprietá.**

Usare questo evento solo su quelle proprietá che non vengono rilevate da ngOnChanges! Forzare un doppio controllo (ed una eventuale modifica duplice dello stato) puó causare comportamenti inattesi dell'applicazione.

ngDocheck viene invocato al termine di ciascun **digest cycle** dell'applicativo.

ngular – ngAfterContentInit / Checked

Sono entrambi **hook specifici dei componenti** poiché riguardano strettamente il **contenuto** che il componente **inserisce nella View.**

Nel caso dell'evento **ngAfterContentInit,** l'hook viene richiamato a seguito dell'inserimento del **contenuto proveniente dall'esterno** (come ad esempio, i parametri in input).

L'hook **ngAfterContentChecked** invece viene richiamato quando il contenuto interno al selettore è stato **proiettato nella View.** Analizziamo cosa si intende per contenuto «proiettato»…

ngular – ngAfterContent: proiezione

Per **proiezione** di contenuto all'interno di una View si intende una porzione di informazione inserita all'interno del selettore di uno specifico componente. Si prenda ad esempio il caso in cui si deve inserire, nel componente, una stringa personalizzata: una soluzione potrebbe essere quella di sfruttare una **proprietá in Input.**

Con il sistema della proiezione invece si inserisce il contenuto personalizzato direttamente all'interno del selettore del componente che deve ricevere delle informazioni. Ad esempio:

```
<selettore><h4>Un titolo proiettato</h4></selettore>
```

ngular – ngAfterContent: proiezione

Il contenuto proiettato nel selettore viene mostrato all'interno della View in corrispondenza di uno specifico tag:

```
<ng-content></ng-content>
```

L'hook ngAfterContentChecked verrá quindi richiamato quando il tag <ng-content/> sará sostituito dal corrispondente contenuto proiettato (nel caso dell'esempio, il titolo nel tag h4).

ngular – ngAfterViewInit / Checked

Anche questi sono **hook specifici dei componenti** poiché riguardano strettamente il **contenuto** che il componente **inserisce nella View.**

Nel caso dell'evento **ngAfterViewInit,** l'hook viene richiamato a seguito dell'inserimento, nei child component, del **contenuto proveniente dall'esterno** (come ad esempio, i parametri in input).

L'hook **ngAfterViewChecked** invece viene richiamato quando il contenuto interno al selettore dei child component è stato **proiettato nella View.**

ngular – ngOnDestroy

L’ultimo hook nella sequenza del Lifecycle di Angular. Viene richiamato subito prima della distruzione del componente (ricorda, JavaScript / Typescript non implementano il concetto di «distruttore»).

Per evitare che alcune risorse rimangano in memoria per un tempo prolungato (il garbage collector di JavaScript potrebbe liberare la memoria occupata da un componente in disuso dopo molto tempo) è fondamentale utilizzare questo hook per **eseguire gli unsubscribe sugli Observable** e **deregistrare gli event handler.**

ngular – ngDoBootstrap

Angular7 introduce un nuovo **hook del lifecycle** dedicato all'istanza dei moduli. Prende il nome **DoBootstrap** e permette di eseguire manualmente la procedura di bootstrap dell'applicativo ignorando il contenuto dell'array *bootstrap* del decorator *NgModule*.

```
@NgModule({
    // Module setup here
})
class AppModule implements DoBootstrap {
    ngDoBootstrap(appRef: ApplicationRef) {
        appRef.bootstrap(AppComponent);
    }
}
```

Grazie a questo hook è possibile stabilire a *runtime* il componente da caricare, sganciandosi dalla tradizionale logica statica del decorator.

6. Template e Data Binding

ngular – Template

Il linguaggio scelto per i Template in Angular è l'HTML. All'interno dei template è **vietato** l'utilizzo del tag script per evitare qualsiasi forma di attacco basato su **injection**. È possibile **estendere** il DOM dell'HTML attraverso la dichiarazione di nuovi **componenti** e **direttive.**

ngular – Template: interpolazione

L'interpolazione permette di gestire il **flusso unidirezionale** dal componente al template. Viene dichiarato attraverso l'utilizzo delle **doppie parentesi graffe** e contiene, solitamente, il nome di una proprietá dichiarata all'interno del componente.

Quanto definito all'interno delle **curly brackets** prende il nome di **Template Syntax:** Angular esegue l'evaluation dell'espressione e ne **converte il valore in stringa.**

{{ 1 + 1 }} => 2

ngular – Template expressions

Una **Template Expression** produce un **valore.** Sono un concreto esempio di Template Expression alcuni tipi di **binding** come l'**interpolazione** ed il **property binding.** Sono espressioni vietate all'interno delle Template Expressions:

- Assegnazioni (=, +=, -=)
- Istanze classi (new)
- Chaining expression (espressioni multiple separate da ; o ,)
- Operatori di incremento e decremento

ngular – Template expressions: contesto

Le Template Expression vengono eseguite nell'ambito dell'istanza del Componente associato al template che le esegue. Altri contesti in cui possono essere valutate, sono:

- Template Input Variable
- Template Reference Variable

Le Template Expressions **non possono accedere al global namespace**: non hanno quindi scope su variabili come window o document, od a metodi come console.log.

ngular – Template expressions

Flow unidirezionale

- Utilizzare una template expression non deve **mai** causare, a seguito della lettura di un dato, il cambiamento di stato nell'applicazione

Change Detection

- Le Template Expressions vengono eseguite ogni qual volta viene rilevato un cambiamento nello stato dell'applicazione attraverso la **change detection strategy**
- Il **change detection cycle** avviene a seguito di una serie di operazioni asincrone come risoluzione delle Promise, risposte Http, pressioni di tasti e movimenti del mouse
- Le espressioni devono essere eseguite in tempi rapidi, pena il verificarsi di rallentamenti

Semplicitá ed Idempotenza

- Le Template Expressions possono anche essere scritte in forma complessa. Bisogna comunque prediligere espressioni brevi e semplici, delegando eventualmente il componente all'esecuzione della logica
- Una espressione ideale é un'espressione **idempotente**, ovvero un'espressione che nel tempo non cambia

Le Template Statements rispondono ad un **evento** emesso da un **elemento target.** Un esempio di **Template Statement** consiste nella porzione destra dell'**event binding:**

```
(event)="method()"
```

Quanto presente dopo il simbolo = viene eseguito, infatti, a seguito dell'intercettazione dell'evento inserito tra le parentesi tonde (a sinistra del simbolo di uguaglianza). A differenza delle Template Expressions, le **Template Statements** hanno un side effect! Non a caso gli eventi vengono intercettati per **modificare lo stato dell'applicativo** in seguito ad una particolare azione dell'utente.

ngular – Template statements

Le Template Statements inoltre permettono l'utilizzo di:

- Operatore di assegnazione (solo =)
- Chaining expressions (; e ,)

Sono invece vietate:

- Istanze classi (new)
- Operatori di assegnazione += e -=
- Operatori di incremento e decremento
- Bitwise Operator

ngular – Data Binding

One Way **data source -> template**

- Interpolazione {{ … }}
- Property binding [target]=“espressione”

One way **template -> data source**

- Event binding (evento)=“statement”

Two way

- Banana in a Box [(target)]=“espressione”

ngular – Two way data binding

Il **two way** data binding si utilizza quando si vuole sia visualizzare una proprietá che aggiornarne lo stato al verificarsi di un cambiamento. La sintassi è una **combinazione** del **property binding** e dell'**event binding.** Per ricordarla, è sufficiente pensare ad una banana () in una scatola []

Una **direttiva** Angular che implementa il two way data binding é **ngModel…**

```
<input [(ngModel)]="username">
<p>ciao {{ username }}</p>
```

Equivale, per la definizione data pocanzi, a questa sintassi:

```
<input [value]="username" (input)="username = $event.target.value">
```

ngular – *ngIf

ngIf è la **struttura di controllo** dei Template di Angular. Permette di eseguire il test di una condizione e mostrare l'elemento se e solo se la data condizione è verificata. Una condizione si dice verificata quando la sua evaluation restituisce **true**.

```
<app-ingredient *ngIf="hasVisibility"></app-ingredient>
```

La struttura di controllo puó essere utilizzata per evitare di accedere ad una proprietá prima che questa sia resa disponibile nel template (ovvero che abbia ancora valore null o undefined). Quando la proprietá sará disponibile e l'evaluation della condizione cambierá, l'informazione verrá mostrata automaticamente.

ATTENZIONE: quando un elemento non supera la condizione espressa nella struttura di controllo, questo **non viene inserito** all'interno del DOM. È cosa ben diversa, ad esempio, rispetto al nascondere l'elemento attraverso i CSS.

ngular – *ngFor

ngFor è la **struttura di iterazione** dei Template di Angular. Permette di ripetere un elemento e tutto il contenuto in lui annidato per ciascun elemento di un array.

```
<app-ingredient *ngFor="let elem of ingredients"
[ingredient]="elem"></app-ingredient>
```

É possibile accedere all'indice relative all'iterazione corrente semplicemente dichiarando una variabile apposita che prenda il valore della proprietá index:

```
<app-ingredient *ngFor="let elem of ingredients; let idx = index"
[ingredient]="elem"></app-ingredient>
```

ngular – ngSwitch

ngSwitch è assimilabile allo *switch statement* di JavaScript per quel che concerne la sintassi e viene impiegato, in Angular, per mostrare un elemento a partire dallo stato di una proprietà. Si articola in 3 direttive che cooperano tra loro:

- **ngSwitch** – determina la proprietà sulla quale valutare I diversi **case**
- **ngSwitchCase** – definisce il valore che se contenuto nella proprietà "target" dello switch, causerà l'inserimento del selettore nel DOM
- **ngSwitchDefault** – L'elemento da inserire nel DOM qualora nessuna condizione espressa nei case sia verificata

```
<div [ngSwitch]="property">
    <app-selector-one *ngSwitchCase="'value-1'"></app-selector-one>
    <app-selector-two *ngSwitchCase="'value-2'"></app-selector-two>
    <app-selector-three *ngSwitchCase="'value-3'"></app-selector-three>
    <app-selector-default *ngSwitchDefault></app-selector-default>
</div>
```

ngular – Reference Template Variable

Un parent controller non puó utilizzare il data binding per invocare I metodi di un componente child o accedere direttamente alle sue proprietá. Si impiegano le **Reference Template Variable:**

```
<app-ingredient #ing></app-ingredient>
```

Se il componente IngredientComponent avesse ad esempio un metodo chiamato myMethod() potremmo accederci, dallo stesso template in cui abbiamo dichiarato la template variable #ing, in un modo analogo al seguente:

```
<app-ingredient #ing (click)="ing.myMethod()"></app-ingredient>
```

Questo sistema é utile quando il componente child non mette a disposizione un set di parametri idoneo a comunicare in modo ottimale attraverso il sistema di binding. Un altro scenario in cui le Reference Template Variable giocano un ruolo fondamentale si verifica quando ci si deve **interfacciare necessariamente a dei metodi del child component** per gestirne il controllo (ad. es. lo start e lo stop di un timer).

ngular – *ngIf - Else

Nella condizione espressa da **ngIf* è possibile definire una condizione alternativa, determinata attraverso la parola chiave *else...*

```
<app-ingredient *ngIf="hasVisibility else other"></app-ingredient>

<app-other #other></app-other>
```

other è un riferimento ad una **reference template variable** associata ad un elemento all'interno del codice che verrà **aggiunto o rimosso** dipendentemente dal risultato della condizione espressa nella direttiva **ngIf.**

*ngIf	Selettore con direttiva	Selettore con RTV
True	Presente	Non presente
False	Non presente	Presente

Angular

7. Routing

In una **app single page** il **router** é delegato all'interpretazione dell'URL presente nella barra degli indirizzi del browser per mostrare una view renderizzata dal client. Ovviamente, la URL puó contenere uno o piú parametri finalizzati alla selezione di una specifica risorsa.

Il router puó, ad esempio, essere agganciato tramite il **binding** ad un link presente nel template (come ad esempio un menu di navigazione).

ngular – Routing: configurazione

Il primo passo per impostare il Sistema di routing di Angular consiste nell'impostare il base path rispetto al webserver. Se l'applicativo non gira in particolari sottocartelle, basterá inserire all'interno del file index.html una dichiarazione come segue:

```
<base href="/">
```

Il router é un pacchetto **esterno ad Angular core.** É raccolto in una libreria dedicate, @angular/router appunto. Per utilizzarlo lo importeremo come un qualsiasi altro pacchetto npm all'interno dell'AppModule:

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

Una app Angular con funzionalitá di routing contiene una istanza **singleton** del servizio Router. Quando la URL cambia, il servizio Router cerca per un indirizzo corrispondente ad un Componente da mostrare.

ngular – Routing: configurazione

src/app/app.module.ts (excerpt)

COPY CODE

```
const appRoutes: Routes = [
  { path: 'crisis-center', component: CrisisListComponent },
  { path: 'hero/:id',      component: HeroDetailComponent },
  {
    path: 'heroes',
    component: HeroListComponent,
    data: { title: 'Heroes List' }
  },
  { path: '',
    redirectTo: '/heroes',
    pathMatch: 'full'
  },
  { path: '**', component: PageNotFoundComponent }
];

@NgModule({
  imports: [
    RouterModule.forRoot(appRoutes)
    // other imports here
  ],
  ...
})
export class AppModule { }
```

ngular – Routing: configurazione

Alcune considerazioni da annotare per l'oggetto Router:

- La keyword data é un oggetto che puó contenere dati arbitrary relative ad una specifica URL. I dati sono accessibili nel momento in cui il router diviene attivo: é il posto ideale per memorizzare dati static come **titoli** e **breadcrumbs**.
- Un path vuoto corrisponde al path di default per l'applicazione
- ** indica il fallback per tutti gli indirizzi che non hanno un match nelle definizioni del router

ngular – Routing: router outlet

Una volta terminata la configurazione del Router, é necessario indicare all'interno del template **dove renderizzare I component soggetti al routing.** Viene in aiuto il **router outlet** ovvero una direttiva particolare che funziona da placeholder per il contenuto che lavora tramite routing.

Quando il router individua un match tra la URL del browser e la URL di un record presente nel Router, renderizzerá il contenuto **subito dopo l'element router outlet.**

```
<router-outlet></router-outlet>
```

ngular – Routing: router links

Individuato il punto in cui renderizzare il contenuto richiesto attraverso il router, non dobbiamo fare altro che **aggiungere dei link di navigazione**. Per creare un link interpretabile dal router useremo la keyword **routerLink** in sostituzione del parametron href.

```
<a routerLink="/recipes" routerLinkActive="active"></a>
```

Il Router permette di associare una classe al link attivo mediante la proprietá **routerLinkActive.** Inoltre, all'interno dell'applicazione, possiamo accedere allo stato del Router attraverso l'oggetto **RouterState.**

ngular – Routing: router state

Al termine di ciascun lifecycle di navigazione, il **router** costruisce una serie di oggetti *ActivatedRoute* che concorrono a definire lo stato attuale dell router stesso. Lo *state* del router é accessibile attraverso la proprietá *routerState* del servizio *Router*.

Ciascun elemento *ActivatedRoute* all'interno del *RouterState* eroga metodi per percorrere a ritroso o in avanti il router state, oltre che fornire informazioni utili ai routes padre, figli e di pari livello.

ngular – Routing: parametri

I parametri indicati attraverso un route sono accessibili attraverso un servizio disponibile nella Dependency Injection: **ActivatedRoute**. Eroga un set di informazioni particolarmente utili sul **route corrente**. Ad esempio

- url – un array di stringhe contenente le singole porzioni del path del route
- data – I dati caricati attraverso le *resolve guard*
- paramMap – una map dei parametri del route, siano essi required o opzionali
- queryParamMap – una map dei parametri esposti nella query del route
- etc...

Per garantire la *retrocompatibilitá* tra Angular 4/5 e la versione 2, rimangono disponibili le proprietá *params* (assimilabile a paramMap) e *queryParams* (assimilabile a queryParamMap). Queste proprietá sono **Observable!** Per poterne analizzare il contenuto si dovrá eseguire una **subscription** all'observable per poter analizzare I dati solo quando questi saranno effettivamente disponibili…

ngular – Routing: parametri

Durante il processo di navigazione, il **Router** emette una serie di eventi attraverso la proprietá *Router.events,* collocati tra l'inizio della navigazione ed il termine della stessa. Ecco un elenco degli eventi emessi:

- NavigationStart – Inizio processo di navigazione
- RoutesRecognized – Percorso riconosciuto nella configurazione del router
- RouteConfigLoadStart – Precede il lazy-load della configurazione del route
- RouteConfigLoadEnd – Segue il lazy-load della configurazione del route
- NavigationEnd – Termine della navigazione
- NavigationCancel – Navigazione annullata (una Guard impedisce l'accesso)
- NavigationError – Navigazione fallita a seguito di un errore

ngular – Routing: guards

Gli strumenti definiti fino ad ora nell'ambito del Routing non permettono di impostare dei criteri che limitino l'accesso di uno specifico set di risorse agli utenti non autorizzati. Vediamo alcuni casi in cui é opportuno limitare la navigazione ad un utente:

- L'utente non é autorizzato alla navigazione (ruolo)
- Non é stata eseguita una procedura di autenticazione (login)
- Devono essere prelevati dei dati prima che il component "target" venga mostrato
- Si vuole richiedere all'utente se I cambiamenti "appesi" (non ancora elaborati) devono essere ignorati o eventualmente salvati in qualche modo

ngular – Routing: guards

Le **Guards** permettono di risolvere queste problematiche aggiungendosi alla configurazione del Route. Il valore restituito da una Guard determina il comportamento della navigazione:

- **True:** la navigazione prosegue
- **False:** la navigazione viene interrotta

Le Guard possono eventualmente imporre al Router di navigare in un'altro percorso, annullando di fatto la navigazione richiesta attraverso l'interazione utente. Sebbene le Guard possano restituire il dato in modo *sincrono,* ci sono dei casi in cui é necessario attendere, ad esempio, la convalida di un webservice (e quindi l'esecuzione di un processo *asincrono*. Per risolvere questo tipo di scenario, le Guard supportano il return di **Observable** e **Promise** di data type *boolean.* Quando viene restituito un Observable o una Promise, la Guard attenderá che la richiesta venga risolta con *true* o *false.*

ngular – Routing: guards

Il Router supporta un set di diverse *interfaces* nell'ambito delle Guard…

- **canActivate** per validare la navigazione ad un route
- **canActivateChild** per validare la navigazione ad un route figlio
- **canDeactivate** per validare la navigazione al di fuori del route corrente
- **resolve** per recuperare dei dati *prima* dell'attivazione del route
- **canLoad** per validare la navigazione verso un modulo caricato in modo asincrono

Le Guards sono **servizi** e pertanto vengono inserite all'interno della Dependency Injection di Angular.

ngular – CanActivate

CanActivate é la Guard delegata alla restrizione dell'accesso utente in base ai privilegi che questo detiene. Ad esempio, é possible permettere l'accesso ai soli utenti che hanno eseguito il login e che appartengono eventualmente ad un *ruolo* specifico.

src/app/auth-guard.service.ts (excerpt)

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

@Injectable()
export class AuthGuard implements CanActivate {
  canActivate() {
    console.log('AuthGuard#canActivate called');
    return true;
  }
}
```

ngular – CanActivate

CanActivate é la Guard delegata alla restrizione dell'accesso utente in base ai privilegi che questo detiene. Ad esempio, é possible permettere l'accesso ai soli utenti che hanno eseguito il login e che appartengono eventualmente ad un *ruolo* specifico.

```
import { AuthGuard }                from '../auth-guard.service';

const adminRoutes: Routes = [
  {
    path: 'admin',
    component: AdminComponent,
    canActivate: [AuthGuard],
```

ngular – CanActivateChild

CanActivateChild, in modo simile alla guard CanActivateChild, é delegate al controllo dell'accesso utente **per I child routes.** Spesso la logica delle guard **CanActivate** e **CanActivateChild** é condivisa: é pertanto consigliato richiamare *CanActivate* all'interno di *CanActivateChild*.

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

  canActivate(route: ActivatedRouteSnapshot, state: RouterStateSnapshot): boolean {
    let url: string = state.url;

    return this.checkLogin(url);
  }

  canActivateChild(route: ActivatedRouteSnapshot, state: RouterStateSnapshot): boolean {
    return this.canActivate(route, state);
  }

/* . . . */
}
```

Angular

8. Form

I **Form** sono un elemento HTML utilizzato per collezionare **dati in input.** Sono una parte fondamentale di qualsiasi applicativo web poiché sono strettamente legati alla **business logic**. Angular permette l'implementazione di form in due diversi approcci:

- Template Driven Form
- Model Driven Form (o Reactive Driven Form)

Sebbene entrambe le metodologie conducano al medesimo risultato, bisogna tener conto delle differenze tecniche dei due sistemi…

Template Driven

- Asincrono
- Generato tramite Direttive

Reactive Driven

- Sincrono
- Generato tramite codice (Form Builder)

ngular – Template Driven Form

Il Sistema **Template Driven** basa la generazione di Form attraverso la scrittura di **template** con sintassi Angular sfruttando delle specifiche **direttive**. La definizione di un Form si articola in punti:

1. Creazione di un **Model**
2. Definizione di un **Component** che controlli il Form
3. Scrittura di un **template** che definisca la struttura del Form
4. Binding delle proprietá del Model a quelle del Form attraverso il **two-way data binding**
5. Aggiunta di un **name attribute** per ciascun Input Field

ngular – Template Driven Form: il Model

Nell'approccio Template Driven, il **Model** é delegato a mantenere lo stato delle proprietá degli Input Fields del Form. Prima di definire un template (e quindi il Form in esso contenuto) é fondamentale avere ben definita la struttura dati che si vuole rappresentare.

src/app/hero.ts

```
export class User {

  constructor(
    public id: number,
    public firstName: string,
    public lastName: string,
    public email string,
    public password string,
    public nickname?: string,
  ) {  }

}
```

La sugar syntax di TypeScript creerá automaticamente delle proprietá di tipo public valorizzandole con I valori passata al costruttore. **Attenzione alla notation del costruttore!** Il simbolo ? permette di definire un parametro opzionale: questo significa che potrá essere omesso nella generazione di un'istanza dell'oggetto.

RICORDA

I parametri opzionali devono essere definiti **per ultimi**

ngular – Template Driven Form: il Component

Un Form Angular é composto da una parte HTML (il **template**) ed il Component ad esso associato che gestisce la user-interaction e lo scambio dati tramite **binding.**

src/app/hero-form.component.ts (v1)

COPY CODE

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

import { User }    from './user';

@Component({
  selector: 'user-form',
  templateUrl: './user-form.component.html'
})
export class UserFormComponent {

  model = new User(null, '', '', '', '');

  submitted = false;

  onSubmit() { this.submitted = true; }

  // TODO: Remove this when we're done
  get diagnostic() { return JSON.stringify(this.model); }
}
```

ngular – Template Driven Form: dipendenze

Affinché I Form template-driven funzionino correttamente é necessario aggiungere le opportune dipendenze (moduli) nel file **app.module.ts**. Per farlo:

1. Importare Il modulo *FormsModule*
2. Includere *FormsModule* nell'array "declarations"

Se un componente, una direttiva o una pipe appartengono ad un modulo dichiarato nell'array imports, **non** é necessario ridichiarare gli elementi nell'array "declarations"!

ngular – il Template

Il template di un Form Angular é composto principalmente da HTML5. Requisito fondamentale per il funzionamento del Form é la presenza dell'attributo *name* su **ciascun tag input**.

I singoli Input Fields devono essere, inoltre, legati alle rispettive proprietá del Model attraverso il **two-way data binding.**

```
<input type="text" id="firstName" [(ngModel)]="model.firstName"
name="firstName">
```

ngular – Template Driven Form: ngForm

La direttiva *ngForm* offre molto piú del two-way data binding: emette informazioni su eventi particolari come ad esempio:

- Interazione sull'input field *(ng-touched)*
- Cambiamento del valore *(ng-changed)*
- Invaliditá del valore *(ng-invalid)*

Questi eventi sono particolarmente interessanti poiché aggiungono delle classi CSS all'input field dipendentemente dallo stato in cui questo si trova. Modificare I CSS di queste classi (es. ng-valid) permette di modificare la UI del Form.

ngular – Template Driven Form: ngSubmit

Dopo aver compilato il Form, l'utente deve inviare I dati inseriti. In Angular il Submit di un Form é un particolare evento chiamato *ngSubmit* sul quale é possibile rimanere in ascolto.

```
<form (ngSubmit)="onSubmit()" #userForm="ngForm">
```

La direttiva ngForm permette di aggiungere al Form delle funzionalitá aggiuntive: permette ad esempio di monitorare lo stato degli Input Field e la loro validitá. I Form hanno, cosí come gli Input Fields, una loro proprietá *valid* che é **true** solo se tutte le proprietá del form sono valide.

ngular – Template Driven Form: validazione

La validazione in un Form con approccio Template Driven consiste nell'uso degli *attribute HTML* impiegati nella validazione nativa HTML. Angular implementa delle *specifiche direttive* corrispondenti agli attribute HTML per poter eseguire la validazione degli input fields attraverso delle funzioni definite nel framework.

Le funzioni di validazione vengono eseguite ogni qual volta si verifica un cambiamento all'interno di un input field di un form per poter definire lo stato attuale del form (VALID o INVALID).

Per analizzare lo stato del Form é possibile associare ngModel ad una *local template variable*…

ngular – Template Driven Form: validazione

```
<input id="name" name="name" class="form-control"
       required minlength="4" appForbiddenName="bob"
       [(ngModel)]="hero.name" #name="ngModel" >

<div *ngIf="name.invalid && (name.dirty || name.touched)"
     class="alert alert-danger">

  <div *ngIf="name.errors.required">
    Name is required.
  </div>
  <div *ngIf="name.errors.minlength">
    Name must be at least 4 characters long.
  </div>
  <div *ngIf="name.errors.forbiddenName">
    Name cannot be Bob.
  </div>

</div>
```

ngular – Template Driven Form: validazione

É importante notare come:

- Gli elementi HTML <input> definiscono le regole di validazione come *required* e *minlength. appForbiddenName* é una direttiva personalizzata poiché non esiste nello standard HTML5
- Associando il valore di ngModel all'interno di una *variabile locale* possiamo riflettere le proprietá del FormControl indicato esattamente all'interno della variabile in questione. Questo permette di controllare le diverse property definite nell'AbstractControl (valid, dirty etc…)
- Le istruzioni di controllo permettono di valutare lo stato della *variabile locale* per poter mostrare dei messaggi di errore qualora necessario

Controllare le properties ***dirty*** ed ***untouched*** permette di non mostrare errori qualora il form non sia ancora stato modificato dall'utente.

Il secondo metodo offerto da Angular per la creazione di Form é l'approccio **Reactive Form** orientato alla *reactive programming.* I Reactive Form prevedono l'utilizzo di particolari oggetti denominati *Form Control* che vengono associati agli elementi presenti all'interno del template.

Un vantaggio nell'uso dei Reactive Form risiede nella sincronia tra I dati del Template e del Form Control: questo permette di non incorrere nelle problematiche di accesso ai dati che affliggono I form template driven. Inoltre, essendo **form sincroni**, sono piú facili da testare mediante *unit test.*

Il Form viene costruito attraverso l'uso dell'oggetto *FormControl*: il component delegato alla gestione del form dovrá quindi importarlo:

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

Ciascun input field del form dovrá avere una proprietá di tipo *FormControl* dichiarata ed istanziata all'interno del component…

```
firstName = new FormControl();
```

Il costruttore dell'oggetto FormControl accetta 3 argomenti **opzionali**:

1. Valore iniziale dell'Input Field
2. Un array di *Validators*
3. Un array di *Async Validators*

Il framework deve essere messo a conoscenza di quale input field é associato a ciascuna proprietá di tipo FormControl. Questa particolare associazione avviene mediante property binding, in questo modo:

```
<input [formControl]=“propertyName">
```

propertyName é il nome della proprietá di tipo FormControl definita nel Component.

ngular – Reactive Form: dipendenze

I Form Reactive sono implementati in Angular all'interno del modulo *ReactiveFormsModule.* Per poterli utilizzare è quindi necessario aggiornare in modo opoortuno il file *app.module…*

1. Import del modulo ReactiveFormsModule
2. Aggiunta del ReactiveFormsModule all'interno dell'array «imports»

Nota: ReactiveFormsModule è un modulo contenuto nel pacchetto @angular/forms

Oltre all'oggetto FormControl, Angular rende disponibili una serie di oggetti utili alla creazione di Form con approccio Reactive:

- **FormGroup**: traccia valore e validitá di un set di FormControl (o AbstractControl)
- **FormArray**: traccia valore e validitá di un array posizionale di elementi FormControl (o AbstractControl)
- **AbstractControl**: è la classe astratta di FormControl, FormGroup e FormArray

ngular – Reactive Form: gruppi

È possibile gestire gruppi di input field attraverso l'oggetto *FormGroup.* Grazie ad esso è infatti possibile tenere traccia di un set di FormControl (o AbstractControl) determinandone inoltre la loro **validitá complessiva.**

src/app/user.component.ts

```
export class UserComponent {
  userForm = new FormGroup ({
    firstName: new FormControl()
  });
}
```

ngular – Reactive Form: gruppi

Il **Template** deve riflettere le modifiche apportate nel Component:

- Il Form deve essere associato al FormGroup
- Ciascun Input Field deve essere associato al giusto FormControl

src/app/user.component.html

```
<h2>User</h2>
<h3><i>FormControl in a FormGroup</i></h3>
<form [formGroup]="userForm" novalidate>
  <div>
    <label>Name:
      <input formControlName="firstName">
    </label>
  </div>
</form>
```

ngular – Reactive Form: builder

Il FormBuilder è un oggetto che permette la creazione facilitata di FormGroup e FormControl. È una soluzione particolarmente conveniente poiché è disponibile come un **servizio** nella **Dependency Injection** di Angular.

src/app/user.component.ts

```
export class UserComponent {
  userForm: FormGroup;

  constructor(private _fb: FormBuilder) {
    this.createForm();
  }

  createForm() {
    this.userForm = this._fb.group({
      firstName: '',
    });
  }
}
```

I Reactive Forms mettono a disposizione due differenti tipi di funzioni delegate alla validazione:

Sincrone: prendono l'istanza del *form control* specificato, ne valutano lo stato e ritornano immediatamente un set di errori oppure null. Vengono definiti come secondo parametro dell'oggetto FormControl

Asincrone: prendono l'istanza del *form control* specificato e restituiscono una *Promise* o un *Observable*che emetteranno un set di errori oppure null. Vengono definiti come terzo parametro dell'oggetto FormControl

Per questioni di performance, Angular esegue la validazione attraverso le funzioni asincrone se e solo se sono risultate prima valide tutte le validazioni sincrone.

ngular – Reactive Form: validazione

Validator è il componente Angular delegato alla validazione dei singoli FormControl. Permette di definire una serie di regole di validazione che consentono di determinare se l'input field è **valido rispetto al suo value.** Alcuni Validators…

- Validators.required
- Validators.maxLength
- Validators.minLength
- Validators.pattern

9. Servizi e Dependency Injection

ngular – Dependency Injection

É un **design pattern** in cui un oggetto riceve le sue dipendenze da un **apposito contenitore di istanze** identificato come **container.** Angular implementa questo design pattern permettendo di condividere istanze di servizi tra I vari elementi di ciascun modulo.

I servizi, per essere presi in carico dal container di Angular, devono essere dichiarati nell'**AppModule** (o nel proprio modulo di appartenenza) all'interno dell'array **providers.** Per poter iniettare un servizio all'interno di un componente si utilizza una particolare notazione di TypeScript direttamente nel costruttore…

Typescript permette di definire in modo rapido delle proprietá ed inizializzarle attraverso il costruttore. Solitamente, l'inizializzazione di un oggetto avviene in questo modo:

```
class Car {
  constructor(engine:string, tires:string, doors:number){
    this.engine = engine;
    this.tires = tires;
  }
}
```

É possibile utilizzare una sintassi piú concisa specificando la keyword di **access control** direttamente nel costruttore ed il tipo di elemento: questo permetterá di creare una o piú proprietá nella classe, nomenclate e valorizzate esattamente come nella firma del costruttore.

ngular – Injection di Servizi

Ecco un esempio equivalente al precedente ma che sfrutta questa particolare feature di Typescript…

```
class Car {
  constructor(private _engine:string, private _tires:string){
  }
}
```

In Angular I servizi vengono iniettati sfruttando proprio questa particolare sintassi: analizzando il costruttore, Angular é in grado di stabilire quali dipendenze ha un particolare oggetto rispetto agli altri. Se un'istanza del servizio é giá disponibile la fornisce, altrimenti ne crea una e la eroga.

Sebbene Angular condivida l'istanza dei servizi con tutte le parti dell'applicativo, é possibile richiedere una nuova istanza isolata da quella disponibile nel container: basta dichiarare un servizio nell'array Providers del decorator. In questo scenario, il sistema di **Dependency Injection** di Angular assume un aspetto **gerarchico.**

Quando invece I servizi vengono dichiarati esclusivamente nell'AppModule, la DI assume le caratteristiche di una **Dependency Injection orizzontale** (o "flat").

Dichiarare quindi un servizio nei providers di un Component porta a differenze di **scope** e **ciclo di vita** di un servizio rispetto ad una dichiarazione all'interno del Module…

@NgModule	@Component
• Registra i Providers nell'injector principale • Eroga l'istanza del servizio in tutto l'applicativo • L'istanza, se generata, ha un ciclo vitale pari a quello dell'app stessa	• Registra i Providers nell'injector dedicato del Component • Eroga l'istanza del servizio nel componente in cui viene dichiarato e nei suoi figli • L'istanza generata ha un ciclo vitale pari a quello del Component e dei suoi figli (onDestroy)

ngular – Injection di Servizi

I Servizi sono quella particolare parte dell'applicativo caricata all'interno del **Dependency Injection Container** delegata a:

- Interrogazione di interfacce ReST
- Calcoli matematici
- Gestione file
- …etc

Un servizio é una **classe plain javascript** decorata con un decorator specifico: @Injectable. Questo decorator serve a comunicare ad Angular che la classe é **iniettabile** nel container. Ai servizi é demandata tutta la gestione della *business logic* dell'applicativo.

ngular – Injection di Servizi

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

import { Recipe } from './recipe';
import {RECIPES} from './mock-recipes';

@Injectable()
export class RecipeService {
  getRecipes(): Recipe[] {
    return RECIPES;
  }
}
```

Angular

10. Observable

ngular – Observable

Gli **Observable** sono un facile strumento di comunicazione tra una risorsa che *emette dei dati* detta **publisher** ed una o piú risorse riceventi denominate **observers.** Gli Observable sono…

Dichiarativi

La callback dichiarata nell'observable viene eseguita solo se esiste almeno un subscriber in ascolto

Indipendenti dal return type

Possono restituire piú valori di qualsiasi tipo, siano essi *literals, messaggi, eventi* etc…

ngular – Creazione di un Observable

Il **publisher** crea un Observable che definisce una function denominata *subscriber.* Questa callback viene eseguita solo quando un observer rimane in ascolto all'Observable stesso.

```
let observable = Rx.Observable.create(function (observer) {
  observer.next(1);
  observer.next(2);
  observer.next(3);
  setTimeout(() => {
    observer.next(4);
    observer.complete();
  }, 1000);
});

console.log('prima del subscribe');
observable.subscribe({
  next: x => console.log('nuovo valore ricevuto: ' + x),
  error: err => console.error('errore: ' + err),
  complete: () => console.log('processo terminato'),
});
console.log('dopo il subscribe');
```

Qualora l'utilizzo degli Observable risultasse complesso, é possibile **convertire un Observable** in una Promise con il metodo .toPromise();

Solitamente un oggetto Promise é piú che sufficiente a soddisfare le esigenze applicative del software. Lavorare con gli Observable é preferibile per la loro grande varietá di ambiti applicativi (ad es. cattura di stream di dati con cadenza arbitraria).

ngular – Leggere dati

Quando un metodo ritorna un oggetto di tipo Observable, é possibile **iscriversi** ad. Ogni observable implementa il metodo subscribe, che notifica la volontá di ricevere dati nel momento in cui questi vengono emesis dal metodo *subscriber.*

```
this._recipeService.getRecipes().subscribe(res => {
   console.log(res);
   this.data = res;
});
```

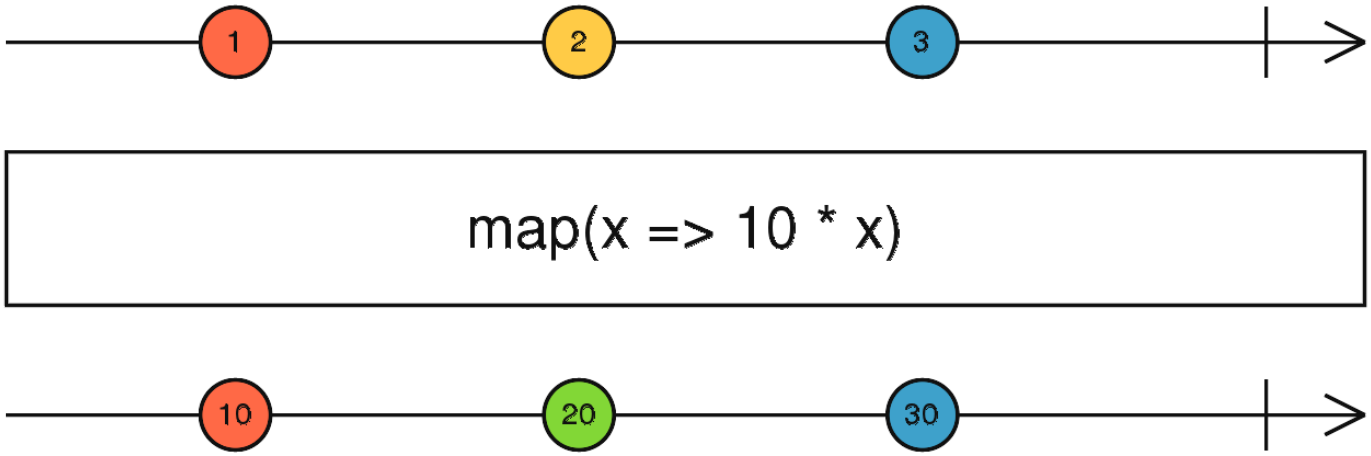
Questi processi sono **asincroni**! Il vantaggio di un servizio asincrono risiede nella possibilitá di eseguire contemporaneamente piú chiamate alla stessa risorsa senza bloccare l'interfaccia utente come avverrebbe con una serie di operazioni sincrone in cascata.

Gli **operators** sono metodi disponibili nel *type Observable,* come ad esempio *map()*, *filter()*, *merge()…* Quando questi vengono invocati ritornano una **nuova istanza** di tipo Observable basata sull'istanza dell'Observable dal quale derivano.

Un *operator* consiste in una *funzione pura* che riceve in input un Observable e restituisce un nuovo Observable in output. Eseguendo un subscribe all'observable in output, si esegue implicitamente un subscribe all'observable in input.

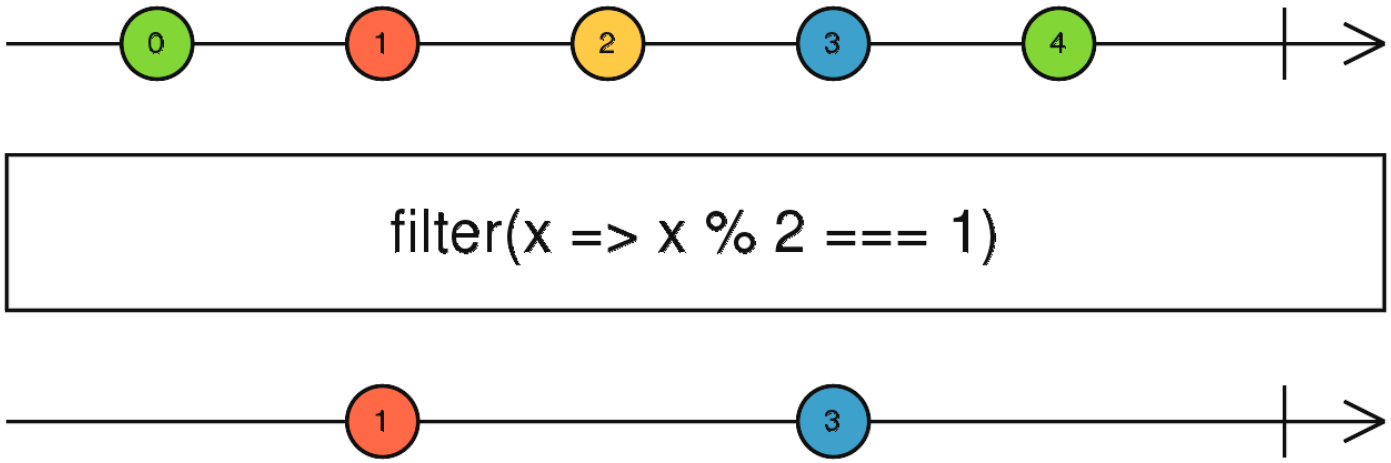
ngular – Operators: map

Cosí come per gli array, il metodo *map* di RxJS applica la callback fornita a ciascun valore emesso dall'Observable.



ngular – Operators: map

Il metodo *filter* permette di ridurre il set di dati emesso dal primo Observable rispetto ad una condizione ben definita. Quando la *callback* fornita restituisce *true* allora il valore corrente viene emesso…



Angular

11. HTTP Module

Le applicazioni, in contesti operative reali, comunicano con servizi esterni per una ben nota serie di operazioni: prime tra tutte, la necessitá di **prelevare dati.** Angular mette a disposizione un modulo dedicato all'interrogazione di servizi esterni tramite protocollo HTTP.

```
import { HttpModule } from '@angular/http';
```

Il modulo HTTP lavora in stretta simbiosi con il pacchetto RxJS: questo significa che, ad esempio, I metodi **get** e **post** ritornano degli **Observable.**

ngular – HTTP ed Observable

```
1.  import { Injectable } from '@angular/core';
2.  import { Http }       from '@angular/http';
3.
4.  import { Observable }     from 'rxjs/Observable';
5.  import 'rxjs/add/operator/map';
6.
7.  import { Hero }           from './hero';
8.
9.  @Injectable()
10. export class HeroSearchService {
11.
12.   constructor(private http: Http) {}
13.
14.   search(term: string): Observable<Hero[]> {
15.     return this.http
16.                .get(`app/heroes/?name=${term}`)
17.                .map(response => response.json().data as Hero[]);
18.   }
19. }
```

Cosí come per il Map operator degli array, il metodo map negli Observable applica una *callback* a ciascun elemento della collection di dati ricevuta. Nell'esempio, ciascuna element della colletion viene convertito mediante casting ad un array di oggetti Hero.

Ricordi? I metodi dichiarati con le Fat Arrow che NON hanno le curly brackets ritornano automaticamente il risultato dell'unica espressione di cui sono composte!

Dalla versione 4, Angular definisce come deprecato il modulo Http analizzato nel capitolo 13 e verrá rimosso definitivamente dalla versione 6. Il Nuovo modulo delegato all'esecuzione di richieste su protocollo HTTP é HttpClientModule…

```
import { HttpClientModule } from '@angular/common/http';
```

Il servizio Http viene invece sostituito da HttpClient.

ngular – Headers

Alcune richieste, per essere esaudite, richiedono l'aggiunta di informazioni all'interno degli *headers.* É il caso, ad esempio, dei *token di autorizzazione* e del *content-type* che descrive il formato con il quale I dati vengono inviati dal client al server.

Angular utilizza un oggetto dedicato alla composizione di headers per le richieste: HttpHeaders. Il costruttore di HttpHeaders accetta in input un oggetto contenente I parametri da aggiungere all'interno dell'header della richiesta in formato *key-value pair*.

ngular – Headers

```
import { HttpHeaders } from '@angular/common/http';

const httpOptions = {
  headers: new HttpHeaders({
    'Content-Type':  'application/json',
    'Authorization': 'my-auth-token'
  })
};
```

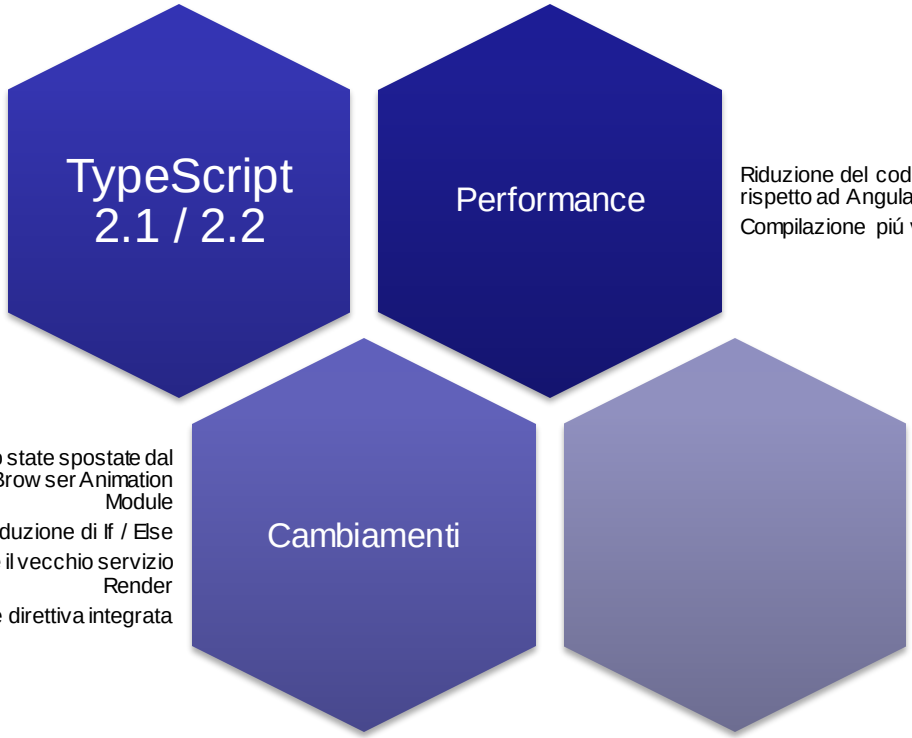
Il servizio HttpClient permette di trasmettere informazioni attraverso I metodi *post, put* e *delete.* Ciascun metodo corrisponde ad una richiesta di tipo HTTP con un http verb ben definito…

Il **secondo parametro** di questi metodi corrisponde ai dati da trasmettere con la richiesta: puó essere un JSON o qualsiasi altro tipo di informazione concordato con l'interfaccia con la quale si vuole interloquire.

Angular

12. Differenze tra versioni

ngular – Versione 4



Riduzione del codice generato di circa il 60% rispetto ad Angular2

Compilazione piú veloce

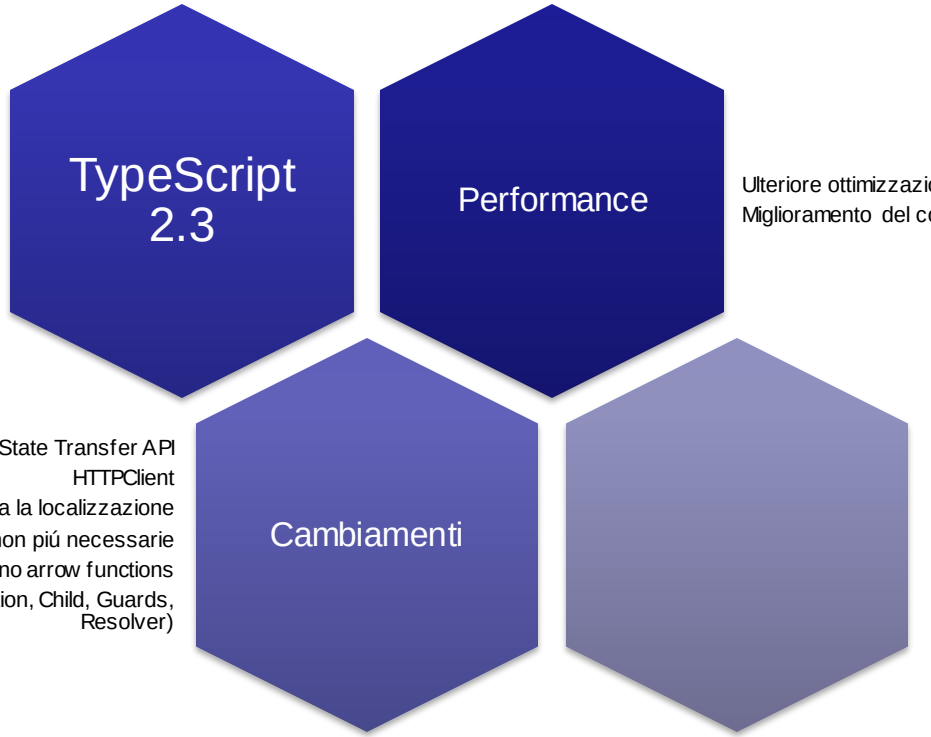
Le **Animations** sono state spostate dal *core* di Angular al Brow ser Animation Module

Introduzione di If / Else

Render2 sostituisce il vecchio servizio Render

Validazione email tramite direttiva integrata

ngular – Versione 5



Ulteriore ottimizzazione della build
Miglioramento del compilatore incrementale

Angular Universal State Transfer API
HTTPClient
Introdotta la localizzazione
Polyfill I18N non piú necessarie
Decorators supportano arrow functions
Router lifecycle (Activation, Child, Guards, Resolver)