

## ANGULAR e IONIC

Rev 2.3 del 21/10/2021

Introduzione ad Angular.js ed Angular 2 .....	2
Angular 2: installazione creazione ed esecuzione di un'applicazione.....	3
Struttura dell'applicazione .....	4
La cartella src .....	5
App Component .....	6
Concetto di Decoratore .....	7
<b>La programmazione</b> .....	7
Binding unidirezionale tramite le doppie graffe .....	7
Creazione manuale di un nuovo componente .....	8
Creazione automatica di un nuovo componente tramite CLI .....	9
Utilizzo del debugger .....	10
Accesso alle risorse statiche .....	10
Le direttive Strutturali .....	11
Le direttive di Attributo .....	12
La gestione degli eventi .....	15
<b>La comunicazione fra componenti</b> .....	16
Accesso agli elementi della pagina html tramite il decoratore @ViewChild .....	16
Accesso ad un bottom component .....	16
Property Binding tramite il decoratore @Input() .....	17
Event Binding tramite il decoratore @Output() .....	18
Ascolto di un evento da codice: il metodo subscribe() .....	19
Utilizzo di bootstrap come libreria interna al progetto .....	20
ngForm and Template Reference .....	20
Direttive personalizzate .....	22
<b>I servizi</b> .....	25
Invio di una richiesta Ajax .....	25
Invio delle richieste ed utilizzo dell'oggetto Observable .....	26
Il controllo dell'applicazione tramite services .....	28
Routing delle pagine .....	30
Routes parametriche .....	33
Child routes .....	34
Lazy Loading Modules .....	37
<b>Pubblicazione di un progetto Angular su firebase</b> .....	38
<b>Creazione di una apk angular cordova</b> .....	40
<b>Cenni su IONIC</b> .....	42
Creazione di un progetto blank .....	42
Creazione di un nuova pagina .....	44
Creazione di un nuovo servizio .....	45
I componenti predefiniti .....	45
Componenti Aggiuntivi : AlertController .....	52
Creazione di un progetto tabs .....	53
<b>Creazione di una apk ionic cordova</b> .....	56

## Introduzione ad Angular

**Angular 1** (noto oggi anche come Angular.js) è una semplice **libreria** javascript nata nell'ambito del progetto **mean** e mirata allo sviluppo del lato client. Fin da subito la principale prerogativa di Angular.js è stata quella di realizzare un **binding bidirezionale** tra una struttura dati in memoria e la rappresentazione della pagina html, evitando così tutte le operazioni di accesso agli elementi della pagina e conseguente caricamento dei dati. Angular.js può essere linkato direttamente nella pagina html utilizzando ad esempio il seguente server CDN:

```
<script
  src="https://ajax.googleapis.com/ajax/libs/angularjs/1.6.9/angular.min.js">
</script>
```

**Dalla versione 2** in avanti Angular è diventato un vero e proprio **framework** per lo sviluppo di applicazioni web lato client. La prima release è stata presentata nel **2016** grazie soprattutto all'impulso di Google. Angular 2 eredita alcuni aspetti base di angular.js ma è profondamente diverso e incompatibile con Angular.js.

### Principali caratteristiche di Angular 2

- Angular2 è completamente centrato sul concetto di **componente** e presenta una spiccata **modularità**, basata sulla possibilità di poter creare nuovi componenti o utilizzare facilmente componenti di terze parti. Ogni componente contiene sia la grafica sia il corrispondente codice di elaborazione, per cui la pagina html risulta estremamente concisa e si limita a richiamare i vari componenti scritti separatamente.
- mentre Angular.js era basato su JavaScript, Angular utilizza un nuovo linguaggio che si chiama **TypeScript** che è una astrazione molto più strutturata rispetto a JavaScript, cioè maggiormente orientato agli oggetti. TypeScript consente un maggiore controllo sul codice. Al momento del build questo codice viene trasformato in js come visualizzato nei messaggi.
- Angular crea al suo interno una vera e propria applicazione che andrà a caricare via via le pagine html quando necessarie. Il comando **ng build** "compila" l'applicazione trasformando i files angular in files html / css / js tradizionali in modo che il browser li possa correttamente interpretare.
- Il comando **ng serve**, esegue un **build temporaneo in memoria** e, in più, avvia l'esecuzione di un server locale nodejs in esecuzione sulla porta 4200 denominato "Angular live development server" che invia al browser le pagine compilate che verranno via via richieste. Inoltre avvia un monitoraggio costante su tutti i file che compongono il progetto. Nel momento in cui si applica una qualsiasi modifica ad un qualsiasi file **l'applicazione viene automaticamente ribuildata** e ricaricata dal browser senza necessità di eseguire un refresh e soprattutto mantenendo lo stato corrente (modalità indicata come **live-reload**).  
Notare che il refresh della pagina è di solito una operazione abbastanza onerosa perché comporti il reinserimento di tutti i dati necessari per arrivare alla pagina di interesse.

### Altre caratteristiche

- E' fortemente orientato al pattern **MVC**
- E' interamente scritto sulla base delle direttive di ECMAScript 2015 (**ES6**)
- E' di tipo **mobile first**, cioè garantisce elevate prestazioni per assicurare una interazione fluida sui dispositivi mobile. Supporta i principali eventi tipici del mobile (es touch e gesture)

Come detto, prima di pubblicare l'applicazione su un server, occorre compilare il codice Type Script in puro JavaScript affinché possa essere interpretato correttamente dal browser.

La fase di compilazione si rende necessaria anche se utilizziamo semplicemente JavaScript arricchito dalle novità di ECMAScript 2015. Salvo specifica configurazione, la compilazione genera codice JavaScript standard in versione **ES5** e garantisce quindi piena compatibilità anche con i browser meno recenti.

**Prerequisiti:** la presenza di node.js (versione uguale o superiore a 10) e npm

**Utilizzo:** Per l'utilizzo di **Angular** esistono delle apposite GUI, ma il modo più diretto è l'utilizzo da linea di comando.

**Add-on** Esistono moltissimi componenti aggiuntivi (add-on) che consentono di estendere angular (griglie, calendari, etc). Ad esempio **Angular Material** è un toolkit che facilita la progettazione in Angular, fornendo una serie di widget già pronti specifici per smartphone.

---

## Installazione di Angular CLI

```
npm install -g ng
```

installa angular come semplice libreria

Viceversa

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

oltre alla libreria, installa anche un Command Line Interface che è un ambiente a riga di comando basato su nodejs per poter creare la struttura di un'applicazione Angular 2 **già configurata** secondo le linee guida ufficiali.

Oltre a consentire la creazione di nuovi progetti, questa CLI è anche molto utile in fase di testing, bundling, e deployment dell'applicazione.

Se l'installazione viene eseguita dal nodejs command prompt, l'installer setta automaticamente anche tutte le variabili di ambiente necessarie.

@angular/cli è ospitato su github, per cui, volendo, è possibile analizzare l'intero codice.

---

## I primi comandi

<b>ng</b>	senza parametri visualizza l'elenco dei comandi disponibili
<b>ng version</b>	restituisce il numero di versione (11.0.5 a dicembre 2020)
<b>ng new</b>	crea una nuova applicazione
<b>ng serve</b>	fa una build e crea al volo un server http per testare la app

L'opzione **--help** dopo ogni comando visualizza le opzioni disponibili per quel comando.

Esempio **ng generate component --help**

---

## Creazione di una nuova applicazione

Guida base disponibile al seguente link: **Angular.io** / GetStarted / Setup

```
ng new ese01
```

Attenzione che nel nome del progetto non sono ammessi spazi o underscore. E' ammesso **solo** il trattino. Questo comando crea una applicazione pura angular (per web, non per smartphone)

Alle varie domande poste digitare **sempre** invio che significa accettare il default.

Le domande sono sostanzialmente tre

- Abilitazione dello Strict mode (**N**)
- Abilitazione dell'Angular routing (**N**)
- Formato del file css (va bene normale **CSS** oppure **SCSS**)

Insieme ad angular/cli vengono installate tutte le dipendenze necessarie. Inoltre viene creato ed inizializzato anche un repository locale di tipo git, compresi i file readme.md, .gitignore, etc)

## Esecuzione dell'applicazione

---

Dalla cartella del progetto lanciare **ng serve --open**

L'opzione **--open** apre automaticamente il browser alla url **http://localhost:4200/** che può essere aperto anche manualmente.

Oltre che in seguito alle modfiiche, l'app viene ricompilata in corrispondenza del comando *save all*

## La pagina DEMO

---

La pagina demo presenta già di per se una serie di link molto interessanti :

- Il primo link apre una applicazione di esempio legata ai supereroi
- Il secondo link apre un manuale della CLI
- Il terzo punta al blog ufficiale di angular

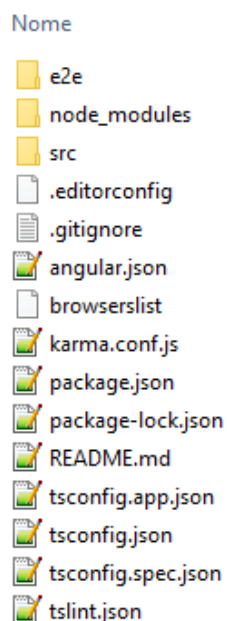
I pulsanti inferiori visualizzano i principali comandi disponibili nella CLI

Se con chrome si ispezionano i sorgenti dell'applicazione, si vede un index.html minimale, e poi c'è una serie di file js molto raffinati che contengono la traduzione del typescript impostato in fase di programmazione.

Nelle build di debug è anche possibile vedere i file typescript associati (web pack / . / src / app)  
Nelle build di release questo non è possibile

## Struttura dell'applicazione

---



**node\_modules** contiene tutti i moduli nodejs necessari per sviluppo ed esecuzione dell'applicazione

**README.md** contiene informazioni generiche su Angular ed un elenco dei passi che occorre fare per avviare l'applicazione, eseguire il build, etc.

**.gitignore** File da non sottoporre a versionig. Attenzione soprattutto a escludere node\_modules

**package.json** contiene :

- il **nome** della app,
- il **numero di versione** impostato come 0.0.0 (release, major number, minor number incrementato in corrispondenza del fix di un problema),
- elenco degli scripts che si possono utilizzare (ng, ng new, ng serve, ng build, etc)
- elenco delle **dipendenze** La principale è sicuramente @angular/core rxjs è il modulo che si occupa della gestione dei servizi rest (chiamate ajax)
- elenco delle **devDependencies**, cioè dipendenze che NON andranno a fare parte della build finale ma che sono necessarie in fase di sviluppo e anche in fase di build.

**angular.json** contiene la configurazione dell'intero progetto e di come dovrà essere eseguito il build.

Tutti questi file in genere non richiedono modifiche.

## La cartella src

E' la cartella che contiene la app vera e propria. I files principali sono:

**index.html** rappresenta la pagina di apertura dell'intera applicazione e contiene nella head i tipici metatag html. Nel body è contenuto un unico **selettore** denominato **<app-root>** e definito all'interno del componente **app.component** Rappresenta il **componente** principale di una applicazione angular. Il file **index.html** va bene così e non deve essere modificato, se non per modificare il **title**

**style.css** contiene gli stili globali dell'applicazione (in realtà gli stili vengono scritti principalmente all'interno dei componenti, per cui questo file conterrà pochi stili generali)

### **main.ts**

rappresenta il vero entry point dell'applicazione, simile al file **program.cs** delle applicazioni C#.

In pratica lancia in esecuzione la classe **AppModule**

Le azioni eseguite all'interno di main.ts sono le seguenti:

- Esegue l'import di due componenti necessari per l'avvio dell'applicazione
- Esegue l'import di **app/app.module.ts** ed **environments/environment.ts** che fanno invece parte dell'applicazione.
  - Il file environments / **environment.ts** è costituito praticamente da una unica istruzione che setta a true/false una variabile denominata **environment.production** che indica se l'applicazione deve essere eseguita in modalità produzione oppure debug. In caso di **environment.production=true** viene eseguita la procedura **enableProdMode()** la quale disabilita tutte le modalità di debugging ed i vari console.log in modo da velocizzare l'esecuzione.
  - Il file app/**app.module.ts** è un file di configurazione che definisce la classe **AppModule** che contiene l'elenco completo di tutti i componenti da caricare per l'esecuzione dell'applicazione
- Lancia l'applicazione vera e propria tramite il comando **platformBrowserDynamic().bootstrapModule(AppModule)**  
Il quale provvede sostanzialmente ad "eseguire" il componente di avvio denominato **AppModule**

### **app.module.ts**

Definisce la classe **AppModule** con all'interno tutti i componenti da caricare per l'esecuzione dell'applicazione. Inizialmente è definito un solo componente, il componente **AppComponent**

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

import { AppComponent } from './app.component';

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

**declarations** contiene l'elenco dei componenti utilizzati all'interno dell'applicazione (al momento soltanto **AppComponent**)

**bootstrap** indica quale componente deve essere utilizzato per l'avvio(**AppComponent**, importato all'inizio da './app.component')

L'ultima riga espone l'intero contenuto di **app.module.ts** come classe di nome **AppModule** utilizzata in **main.ts**

Ogni nuovo componente dovrà preventivamente essere dichiarato all'interno di questo file.

Tutti i file relativi al progetto utente sono nelle sottocartelle **app**

Tutto il resto sono files di configurazione che difficilmente dovranno essere modificati.

## app.component

In angular ogni cosa è un **COMPONENTE**. Un **componente** è una classe (con iniziale maiuscola) con associato un **decoratore** (messo a disposizione dal core) che la 'trasforma' in componente .  
**app.component** è il **componente principale visualizzato da main.ts**.

Associati ad ogni componente ci sono sempre 4 files:

- 1) **app.component.ts** è il file più importante fra i quattro. Rappresenta il file **Type Script** all'interno del quale è definita la classe che realizza il componente:

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'ese01'; // property della classe
}
```

- All'inizio viene importato il decoratore **Component** che è un oggetto definito all'interno della libreria **@angular/core**
- Il **decoratore** **@Component** viene applicato alla classe **AppComponent** aggiungendo nuove caratteristiche alla classe stessa e trasformandola in un **Componente Angular**.

- Il decoratore `atComponent` assegna le tre caratteristiche fondamentali di un componente Angular:
    - definisce un **selettore** che consentirà di utilizzare il componente all'interno di una qualsiasi pagina html.
    - assegna al componente un **file html** associato in cui viene definita la struttura html del componente
    - assegna al componente un **file css** associato inizialmente vuoto.  
In realtà il link al file css è costituito da un vettore enumerativo in quanto è possibile associare ad una stessa classe più files css / scss
  - L'oggetto **AppComponent** viene 'esportato' tramite una istruzione **export (ES6-modules)** **AppComponent** è l'oggetto che viene mandato in esecuzione da **AppModule.ts**
  - All'interno della classe **AppComponent** è possibile definire liberamente nuove Proprietà e nuovi Metodi personalizzati.
- 2) **app.component.html** memorizza il contenuto html relativo al componente. Si può provare ad esempio a cambiare la scritta "Welcome" in alto a sinistra. L'icona di twitter a destra è disegnata manualmente tramite un tag SVG. All'interno del file html si possono inserire normali tag html oppure altri componenti angular creati appositamente. Al fine di migliorare la portabilità è raccomandato l'utilizzo di soli componenti angular.
- 3) **app.component.css** E' la pagina in cui è possibile definire i formati di stile in formato **css/scss**
- 4) **app.component.spec.ts** Facoltativo, consente di definire dei test sull'applicazione

Invece di `templateUrl` e `styleUrls` si possono utilizzare `template` e `style` che, anziché gestire il nome di un file esterno, consentono di scrivere direttamente il codice html interno ed i relativi stili.

## Concetto di Decoratore

Un decoratore, introdotto dal simbolo `@` (at), è un metodo di classe che **consente di aggiungere nuove caratteristiche all'oggetto a cui viene applicato**. Si aspetta come parametro sempre un json che conterrà informazioni diverse a seconda del tipo di decoratore.

Esistono diversi tipi di decoratore:

**@Component()** trasforma la classe in un Componente Angular utilizzabile nelle pagine html.

**@ViewChild()** consente alla classe di accedere agli elementi HTML della pagina

**@Input()** consente ad un Top Component di iniettare un valore in fase di creazione di un Bottom Componente

**@Output()** opposto al precedente. Consente ad un bottom componente di generare un evento che potrà essere intercettato dai top component con la conseguente possibilità di passare dei parametri dal bottom component al top component

**@Injectable()** rende la classe in cui è definito "iniettabile" al costruttore di un qualsiasi componente.

## La programmazione

### Binding unidirezionale all'interno di un testo html tramite le Doppie Graffe

Il file `.html` definisce la grafica della pagina.

- Può far uso di tag html oppure di componenti angular



- Può accedere a Property e Methods delle classe. Tutto ciò che viene dichiarato dentro la classe risulta visibile ed accessibile dall'html.
- L'accesso in lettura alle property della classe associata può essere eseguito tramite l'utilizzo di un **segnaposto** associato ad una Property della classe (**binding unidirezionale**).  
Un segnaposto è definito tramite una coppia di **doppie graffe** all'interno delle quali si inserisce il nome della variabile associata: **{{nomeProperty}}**. Si parla di **interpolazione di stringa**.  
Le doppie graffe **renderizzano** (visualizzano) un contenuto html sulla base del contenuto di una variabile ts.

Note:

- Il nome della variabile all'interno del segnaposto può essere scritto CON o SENZA il **this**.
- Il segnaposto in ogni momento riflette il valore della variabile associata.
- Il segnaposto può essere inserito in qualsiasi punto di una pagina html, sia all'interno di un testo sia all'interno di un qualsiasi tag o attributo.
- Letta al contrario, si dice che la classe valorizza il marcatore **{{nomeVariabile}}** definito all'interno della pagina html. `<span>{{title}} app is running!</span>`

## Creazione manuale di un nuovo componente

Scopo di un componente è quello di poter essere utilizzato all'interno di una pagina html.

All'interno della cartella **src / app** occorre creare una **sottocartella** student.

Ognuna di queste sottocartelle dovrà contenere i 3 seguenti files.

```
student.component.ts
student.component.html
student.component.css
```

- **student.ts**, prendendo spunto dall'app-component, scrivere il seguente codice di dichiarazione ed esposizione del nuovo componente:

```
@Component ({
  selector: 'app-student',
  templateUrl: './student.component.html',
  styleUrls: ['./student.component.css']
})
export class StudentComponent {
  name = 'pippo';
}
```

Una classe, per poter essere visibile dagli altri moduli, deve SEMPRE essere esposta mediante

```
export class ClassName { }
```

- **student.html** si può scrivere normale codice html:

```
<p> Hi, my name is {{name}} </p>
```

I nomi sono arbitrari ma è consigliato mantenere la stessa simbologia usata dal wizard.

Al selettore si assegna sempre un nome del tipo `<app-componentName>`. Si aggiunge app davanti per non interferire con i selettori html (il tag `student` non esiste, ma altri nomi potrebbe già esistere fra i tag html, ad esempio `article`)

**Nel momento in cui si digita @ e si sceglie il decoratore dall'elenco a discesa, automaticamente viene aggiunto in alto l'import corrispondente.**

Il parametro atteso dal decoratore è un json costituito dai tre campi indicati.

Notare che nei valori da assegnare a **templateUrl** e **styleUrls** occorre omettere il nome della cartella student, riportando invece direttamente i nomi dei files.



Le chiavi `templateUrl` e `styleUrls` stanno ad indicare che i valori indicati di seguito contengono la URL di un file. La chiave `template` si aspetta invece come parametro l'intero contenuto del file html interamente racchiuso all'interno del cosiddetto **backtick** (apice singolo rovesciato ALT+96). Es:

```
template: `

hi!  </p>`


```

---

### Utilizzo del componente all'interno di un altro componente

```
<h2>Elenco Studenti</h2>
<app-student>  </app-student>
<app-student>  </app-student>
<app-student>  </app-student>
```

---

### Registrazione del componente all'interno del file app.module.ts

Ogni volta che si crea un nuovo componente occorre "registrarlo" all'interno del file app.module.ts. Le righe da aggiungere sono due:

```
import {StudentComponent} from './student/student.component';

declarations: [
  AppComponent,
  StudentComponent
],
```

Nel momento in cui si scrive `StudentComponent` visualStudioCode provvede automaticamente ad aggiungere l'import.

---

### Creazione automatica di un nuovo componente tramite CLI

Per creare un nuovo componente si può utilizzare la CLI di angular che fa tutto da sola. A tal fine occorre **fermare il server** e digitare il seguente comando:

```
ng generate component student
ng g c student
```

digitando `ng generate --help` viene visualizzato l'elenco dei possibili comandi

Il comando precedente genera i quattro files relativi al componente ed aggiorna automaticamente il file `app.module.ts`

```
ng generate component student --skip-tests
```

L'opzione `--skip-tests` evita la creazione del file di test

Anche omettendo l'opzione `skipTests` il file di test può comunque essere cancellato in qualsiasi momento.

All'interno del file html il wizard aggiunge semplicemente un tag `<p>` esattamente come nella classe precedente: `<p> student works! </p>`

All'interno del file .ts, oltre a ciò che è stato fatto nella classe precedente, viene aggiunto anche un metodo `ngOnInit()` che è una specie di `page_load` cioè una callback richiamata DOPO rispetto al costruttore e che può essere utilizzata per le ultime inizializzazioni prima della visualizzazione del codice html associato.

## Utilizzo del debugger

E' possibile eseguire un **Debug** direttamente sul file typescript. Sulla finestra **Sources** aprire

**webpack** / **.** / **src** / **app** / student / **student.ts**

Se metto un breakpoint nel costruttore ci passerà n volte, una volta per ogni selettore creato nella pagina html principale.

## Accesso alle risorse statiche

Le risorse statiche, come ad esempio le immagini, possono essere salvate all'interno della cartella **src/assets**, come espressamente indicato nel file di configurazione **angular.json** alla riga 22 che definisce dove andare a cercare gli assets (cioè le risorse statiche).

```
"assets": [  
  "src/favicon.ico",  
  "src/assets"  
],
```

Ovviamente questa impostazione può essere modificata, oppure si può inserire un path aggiuntivo all'interno del vettore.

Il path per accedere alle immagini da codice sarà **"assets/nomeFile"**

```

```

## Lettura di un file json

I files json invece possono essere salvati all'interno di un qualsiasi componente (quello che intende farne uso) e si potrà accedere al file.json mediante un normale import:

```
import students from "../students.json"
```

A tal fine occorre però impostare all'interno del file **tsconfig.json** le seguenti configurazioni:

```
"resolveJsonModule": true,          // esattamente come in node js  
"allowSyntheticDefaultImports" : true,
```

## Il file tsconfig.json

Consente interessanti configurazioni relativamente al progetto corrente.

Oltre alle due precedenti, che abilitano angular alla lettura dei file json, sono disponibili:

```
"strictPropertyInitialization": false,  
in caso di valore true (default) typescript richiede l'inizializzazione esplicita di tutte le property.  
Molto scomodo soprattutto nel caso delle classi.
```

## Le Direttive strutturali

Le direttive strutturali consentono di inserire delle istruzioni Angular all'interno dei tag HTML, istruzioni che permettono di modificare il DOM a runtime aggiungendo o togliendo elementi.

Consentono sostanzialmente di devinare la struttura html del componente.

Le direttive strutturali introducono un contesto Angular che verrà poi tradotto in type script, per cui al loro interno si possono utilizzare direttamente le variabili definite all'interno del file type script.

### La direttiva \*ngIf

```
<p>
  Gestione Contatti
  <span *ngIf="nomeUtente"> Contatto: {{nomeUtente}} </span>
  <span *ngIf="nomeUtente!=undefined"> Contatto: {{nomeUtente}} </span>
</p>
```

Le due righe sono equivalenti.

Come valore della direttiva \*ngIf si può assegnare una variabile booleana definita all'interno della classe oppure una condizione. Se la variabile / condizione risulta vera, il tag <span> viene **visualizzato**, altrimenti **non** viene visualizzato.

### La direttiva else

Se la condizione della **ngIf** risulta falsa, si può utilizzare all'interno della IF medesima una **else** tale da visualizzare un template alternativo nel modo seguente:

```
<p *ngIf="serverCreated; else noServer" >
  Server was created. Server name is: {{serverName}}
</p>
<ng-template #noServer>
  <p> Server is being created </p>
</ng-template>
```

Se la variabile `serverCreated` assume il valore true, l'intero template viene rimosso dal DOM.

### Il ciclo \*ngFor

Supponendo di definire all'interno del file ts un vettore enumerativo contatti, si possono visualizzare tutti i contatti all'interno di un contenitore nel modo seguente:

```
<div >
  <p *ngFor="let contatto of contatti; let i = index" >
    <span>{{contatto.nome}} {{contatto.cognome}}, {{contatto.eta}} anni</span>
  </p>
</div>
```

Il ciclo FOR definisce un **cursore** contatto che scorre tutti i contatti presenti nel vettore enumerativo. Ad ogni ciclo **visualizza il tag <p> in cui si trova**, replicando anche tutti i tag interni.

All'interno del ciclo FOR, in coda alle istruzioni canoniche, è possibile definire e valorizzare nuove variabili. All'interno del ciclo FOR la parola chiave **index** rappresenta l'indice del record corrente all'interno della scansione (a base 0)

Il ciclo **ngFor** consente esclusivamente l'iterazione attraverso una collezione o un vettore.

Non è consentito un ciclo diretto del tipo `*ngFor="let i=1; i<20; i++"` che può comunque essere simulato nel modo seguente (orribile) `<p *ngFor="let number of [1, 2, 3, 4, etc.]"`

## Le Direttive di Attributo

Le **Direttive di attributo** consentono di personalizzare il comportamento di un tag o di un componente, collegando direttamente una proprietà di stile o una classe al valore di una variabile definita all'interno del file typescript.

Le più importanti Direttive di Attributo sono:

**[ngStyle]** (Direttiva di Stile)

**[ngClass]** (Direttiva di Classe)

**[(ngModel)]** (Direttiva per il binding bidirezionale)

Le direttive devono sempre essere scritte fra parentesi quadre perchè come valore a destra si aspettano sempre una variabile typescript oppure una funzione.

Davanti al nome della variabile type script si può indifferentemente usare / non usare il **this**

**NOTA:** Le direttive di Angular iniziano sempre con **ng**. Invece di **ng**, per rendere la pagina valida dal punto di vista dell'html, si può utilizzare **data-ng** che è html standard ma un po' troppo prolisso.

### [ngStyle]

Utilizzata per applicare al volo le proprietà di stile agli elementi HTML; **si aspetta come parametro il nome di una proprietà di stile a cui viene assegnato il valore di una certa variabile** (o meglio di una proprietà della classe associata). L'attributo **[ngStyle]** può essere richiamato una sola volta, però al suo interno si possono gestire parallelamente più Proprietà di Stile separate da virgola.

```
<p [ngStyle]="{ 'backgroundColor': student.gender=='F' ? 'pink' : 'cyan',  
               'fontWeight' : nVulnerabilities > 10 ? 'bold' : 'normal' }" >
```

Ad ngStyle possono essere associate una o più proprietà di stile, a cui viene assegnato un valore differente a seconda che la successiva espressione booleana di valorizzazione risulti vera oppure falsa.

- Il primo parametro rappresenta il nome della classe,
- il secondo parametro il criterio di applicazione della classe medesima

Va bene quando ci sono piccole cose da fare al volo.

### [ngClass]

Più strutturata e più elegante rispetto alla precedente. ngClass **si aspetta come parametro il nome di una classe che viene applicata o meno a seconda che la successiva espressione booleana di valorizzazione risulti vera oppure falsa**. L'attributo **[ngClass]** può essere richiamato una sola volta, però al suo interno si possono gestire parallelamente più Classi separate da virgola.

```
<p [ngClass]="{ 'female': student.gender=='F',  
               'male': student.gender=='M',  
               'underlined': student.city=='Fossano' }" >
```

```
<p [ngClass]="{ 'blinking': nVulnerabilities > 10 }" >
```

```
<p [ngClass]="{ 'green': serverStatus=='online', 'red': serverStatus=='offline' }" >
```

- Il primo parametro rappresenta il nome della classe,
- il secondo parametro il criterio di applicazione della classe medesima

**[ngStyle] Soluzione Alternativa**

Invece di inserire il codice all'interno della pagina HTML, una soluzione più strutturata consiste nel creare all'interno della classe TS uno specifico metodo che ritorni il colore di sfondo da applicare al tag <p>

```
getBackColor() {  
    const color:string = this.student.gender=='F' ? 'pink':'cyan';  
    return {backgroundColor : color};  
}
```

All'interno del json di ritorno si possono ovviamente inserire più proprietà.

Il file.html risulterà così molto più leggero:

```
<p [ngStyle]=getBackColor()>  
    {{student.name}} - {{student.city}}  
</p>
```

**Property Binding di un attributo HTML tramite [ ]**

Oltre all'interpolazione di stringa, per assegnare ad un attributo html il valore di una variabile ts si può utilizzare anche il cosiddetto **Property Binding** che consiste nello scrivere il nome della property tra **parentesi quadre** e poi assegnare come valore il nome della variabile da visualizzare.

Le **parentesi quadre** stanno ad indicare che nella stringa di destra non è contenuto un valore diretto, ma una variabile della quale occorre leggere il contenuto.

Le seguenti due sintassi sono praticamente equivalenti:

```
  
<img [src] ="recipe.imagePath">  
  
<input type="text" [value]="recipeName">
```

Il **Property Binding** rispetto alle `{{ }}` è più generale (può essere applicato anche alle classi) e gestisce meglio eventuali eccezioni (ad esempio una immagine che ritarda ad arrivare o non arriva) per cui, a parità di condizioni, risulta preferibile.

**Per gli attributi booleani**, invece di una singola variabile, si può impostare una condizione: se la condizione risulta true l'attributo viene applicato, altrimenti NON viene applicato:

```
<button [disabled]="studentname.length==0" (click)="onCreateStudent()">  
<button [disabled]="studentName==' ' " (click)="onCreateStudent()">
```

Cioè il pulsante risulta abilitato SOLO se la variabile `studentname` risulta diversa da stringa vuota

Il Property Binding, come detto, può essere applicato anche alle classi:

```
<div [class.collapse]="collapsed">
```

Cioè la classe `'collapse'` viene applicata se la variabile `collapsed` è true e viceversa

Attenzione che questa sintassi può essere **SOLO unidirezionale** dal codice verso l'interfaccia.

Cioè l'interfaccia valorizza costantemente il valore della variabile.

**NON può diventare bidirezionale** aggiungendo delle parentesi tonde all'interno delle parentesi quadre. Le parentesi tonde possono anche essere inserite, ma il flusso rimane unidirezionale.

Per poter realizzare un binding **bidirezionale** occorre utilizzare la seguente Direttiva **ngModel**

**[ (ngModel) ]**

**ngModel** consente di eseguire un **binding bidirezionale** tra una variabile TS e la **proprietà predefinita** del controllo corrente (**value** nel caso di un Text Box, **checked** nel caso di un checkbox o radiobutton).

Ad esempio nel caso di un textBox le seguenti righe sono equivalenti:

```
<input type="text" [(value)]="serverName"> // nok  
<input type="text" [(ngModel)]="serverName"> // ok
```

**Omettendo le tonde interne**, il binding diventa **unidirezionale**, dal codice verso l'interfaccia.

Poco utile in quanto diventa equivalente al Property Binding eseguito tramite parentesi quadre [value]

**Note:**

1. Per poter utilizzare [ (ngModel) ] è necessario importare manualmente il componente **FormsModule** all'interno del file **app.module.ts**
2. Nelle ultime versioni, nel momento in cui si utilizza [ (ngModel) ], l'interprete TypeScript genera un **errore** segnalando di aggiungere anche la proprietà  
[ngModelOptions]="{standalone:true}"
3. Se si utilizza **ngModel** all'interno di un tag <form>, è necessario impostare all'interno del tag anche l'attributo **name** che viene utilizzato da Angular per gestire il collegamento tra attributo gestito da ngModel e variabile collegata (vedasi Direttiva **ngForm**).

Altrimenti l'inspector segnalerà il seguente errore:

*if ngModel is used within a form tag, either the name attribute must be set or the form control must be defined as 'standalone' in ngModelOptions.*

**Link del componente FormsModule all'interno del file app.module.ts**

```
import { BrowserModule } from '@angular/platform-browser';  
import { NgModule } from '@angular/core';  
import { FormsModule } from '@angular/forms';  
  
import { AppComponent } from './app.component';  
import { ServerComponent } from './server/server.component';  
import { ServerFarmComponent } from './server-farm/server-farm.component';
```

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

## La gestione degli Eventi

Per quanto riguarda gli eventi, è sufficiente scrivere il nome dell'evento all'interno di parentesi tonde:

```
<button (click)="assegnaNome()"> cliccami </button>
```

In corrispondenza di ogni click verrà richiamata la procedura "assegnaNome()" scritta all'interno del file .ts. Notare che la procedura deve essere scritta come stringa con le tonde finali, esattamente come si fa per richiamare una procedura javascript dall'interno di una pagina html: onClick = "assegnaNome()". Semplice richiamo diretto a procedura.

```
<input type="text" (input)="onChangeName()">
```

L'evento (input) di un textBox viene richiamato ogni volta che si digita un tasto

### Accesso da codice al puntatore dell'oggetto che ha scatenato l'evento

Il **this** non è utilizzabile in quanto gli eventi non sono associati tramite **addEventListener()**, ma sono semplicemente 'richiamati' dall'html. Per accedere dalla procedura di evento al puntatore dell'elemento che ha scatenato l'evento ci sono diverse possibilità:

#### 1) utilizzo di \$event

A tutti gli eventi javascript viene automaticamente iniettato un parametro **event** che al suo interno contiene una proprietà **event.target** che rappresenta il puntatore all'elemento che ha scatenato l'evento. Il parametro **event** non ha altre proprietà particolarmente interessanti per cui, se si ha a disposizione il **this**, il parametro **event** perde di significato. In Angular il passaggio del parametro **event** alla procedura di evento non è automatico ma, in fase di chiamata, occorre utilizzare la parola chiave **\$event** che indica ad Angular di propagare i parametri alla procedura di evento.

La procedura di evento riceverà a questo punto il parametro **event** che potrà rinominare a suo piacimento (ed esempio **e**) ed utilizzare per accedere al parametro **e.target**

Esempio di click su un listBox:

```
<select (change)="visualizzaRecord($event)">

visualizzaRecord(e) {
    console.log(e.target.value) // value della voce selezionata
}
```

#### 2) utilizzo di un segnaposto #

In fase di creazione del tag è possibile salvarsi il puntatore al tag tramite l'utilizzo di un segnaposto #, puntatore che potrà poi essere passato alla procedura di evento.

Riscrittura dell'esempio precedente

```
<select #lstCitta (change)="visualizzaRecords(lstCitta)">

visualizzaRecords (ref) {
    console.log(ref.value) // value della voce selezionata
}
```

#### 3) Passaggio del parametro alla funzione di evento

Trattandosi di un richiamo diretto e non di un semplice puntatore, è possibile passare alla funzione di evento qualsiasi parametro visibile al momento del richiamo:

```
<a *ngFor="let user of users" (click)="onClick(user.id)"> . . . . </a>
```



## La comunicazione fra componenti

### Accesso agli elementi della pagina html

Una funzione di evento talvolta può avere necessità di accedere agli altri elementi della pagina html.

A tale scopo esistono due appositi decoratori che si aspettano come parametro un selettore CSS.

- `@ViewChild` restituisce il primo matching element all'interno della pagina
- `@ViewChildren` restituisce tutti gli elementi individuati come una `QueryList of items`

### Esempio

Definito un ID `"txtName"` all'elemento della pagina a cui si vuole accedere, occorre definire nel file di classe una variabile di tipo `ElementRef` (puntatore a oggetto che andrà a 'wrappare' l'oggetto html) preceduta dal decoratore `@ViewChild`

```
import { Component, ViewChild, ElementRef } from '@angular/core';  
@ViewChild("txtName") _txtName! : ElementRef;
```

cioè definiamo una variabile denominata `_txtName` di tipo `ElementRef` che punta all'elemento HTML avente come `id="txtName"`.

*Il **punto esclamativo** finale in coda alla variabile `_txtName` è un "trucco" per risolvere un errore di sintassi di typescript. In pratica con il punto esclamativo si garantisce a typescript che abbiamo la certezza che esista nell'html un elemento avente come id `txtName`, cioè ancora che la variabile `_txtName` non potrà essere undefined.*

A questo punto in qualunque evento si può utilizzare la variabile `_txtName` seguita dalla property `nativeElement` che consente di accedere all'oggetto html interno e quindi a tutte le sue proprietà..

```
this._txtName.nativeElement.disabled=true;
```

### Accesso ad un bottom component

Sempre tramite `ViewChild` è anche possibile accedere ad un bottom component utilizzato all'interno della pagina. Nell'esempio si accede al primo `StudentComponent` definito nella pagina html:

```
@ViewChild(StudentComponent, {static:false}) _child : StudentComponent;
```

L'opzione `{ static:false }` fa sì che la variabile `_child` sia aggiornata in corrispondenza di ogni variazione sul componente figlio

In alternativa è anche possibile assegnare alle varie istanze del `ChildComponent` un semplice **Template Reference:** (pag 19)

```
<child-component #child></child-component>  
@ViewChild("child", { static: true }) _child: ChildComponent;
```

## Property Binding tramite il decoratore @Input()

Un **Top Component** (che sta sopra), nel momento in cui utilizza all'interno del proprio file html il selettore di un **Bottom Component** (Bottom Component che quindi vivrà all'interno del Top Component), può passare al Bottom Component uno o più parametri definiti al momento stesso dell'utilizzo del selettore.

A tal fine il Top Component deve definire tra parentesi quadre una nuova variabile (in questo caso denominata genericamente [ **nuovoStudente** ]) alla quale, mediante un **Property Binding** andrà ad assegnare il contenuto di una variabile di ciclo oppure di una variabile definita nella classe.

### students

```
<app-student *ngFor="let student of students" [nuovoStudente]="student.name">
</app-student>
```

### student

Il Bottom Component dovrà dichiarare la Property **nuovoStudente** tramite il decoratore @Input() che lo avvisa che questa variabile verrà passata dalla classe dal Top Component in fase di dichiarazione del componente. La variabile **nuovoStudente** potrà poi essere normalmente utilizzata all'interno della classe e della pagina html associata

```
@Input() nuovoStudente:string
```

Si parla di **Property Binding** in quanto la comunicazione avviene mediante il passaggio di una Property dal Top Component al Bottom Component. Il meccanismo del Property Binding permette di assegnare ad una proprietà di un elemento il valore di un'espressione valutata da Angular. Tale proprietà viene identificata col nome di target property.

Attenzione che le variabili ricevute tramite @Input() **NON** sono visibili all'interno del costruttore, ma sono invece visibili all'interno di **ngOnInit()** che viene richiamato DOPO rispetto alla valorizzazione dei parametri di tipo @Input().

## Esempio 2

In quest'esempio il Top Component è una **lista di ricette** che, mediante un ciclo ngFor, genera tanti recipe-item passando come parametro l'intera ricetta che dovrà essere visualizzata all'interno del recipe-item:

### recipe-list

```
<app-recipe-item *ngFor="let item of recipes" [recipe]="item">
</app-recipe-item>
```

### Recipe-item

```
@Input() recipe:RecipeModel
```

## Sequenza completa di caricamento di un componente

Quando un Top Component utilizza nella sua pagina html il selettore di un **Bottom Component** la sequenza di operazioni che vengono eseguite all'interno della classe del Bottom Component è la seguente:

1. Istanza della classe con richiamo del costruttore
2. Valorizzazione delle variabili in input (dichiarate tramite il decoratore @Input())
3. Richiamo dell'evento **onInit()**
4. Visualizzazione del Bottom Component all'interno dell'area del Top Component.

## Event Binding tramite il decoratore @Output()

L'operazione inversa che consente di passare un parametro da un Bottom Component verso un Top Component è decisamente più complessa.

Si parla questa volta di **Event Binding** in quanto la comunicazione avviene attraverso la generazione di un evento custom emesso dal Bottom Component verso il Top Component.

**Il Bottom Component** (nell'esempio la barra di navigazione <app-header>) deve generare un evento custom al quale passa come parametro i valori da inviare al top component (nell'esempio app-component) :

```
<a href="#" (click)="onSelectClick('recipes')">Recipes</a>
<a href="#" (click)="onSelectClick('shopping-list')">Shopping List</a>

// dichiarazione di un evento di tipo EventEmitter a cui passeremo una string
@Output() featureSelectedEvent = new EventEmitter<string>();

// in corrispondenza del click generiamo l'evento e gli passiamo la stringa
onSelectClick(feature: string) {
  // generazione dell'evento
  this.featureSelectedEvent.emit(feature)
}
```

**EventEmitter** è un **Typed Object**, cioè occorre specificare quale o quali tipi di parametri passeremo a questo evento. Nell'esempio si passa una semplice stringa identificativa, ma si potrebbero passare anche **più parametri** o eventualmente **nessuno** (<void>)

L'oggetto **featureSelected** è preceduto dal decoratore **@Output** che fa sì che questo evento possa essere propagato ai vari top component, cioè a quei componenti sovrastanti che utilizzano il componente corrente.

Il metodo **emit()** dell'oggetto **featureSelected** è quello che genera fisicamente l'evento passandogli come parametro il valore feature ricevuto da onSelect()

**Attenzione a scegliere EventEmitter da Angular Core, perché ci sono altri oggetti diversi con lo stesso nome**

**Il top component** (nell'esempio app-component), all'interno dell'html, al momento della dichiarazione del bottom component (<app-header>) deve mettersi in ascolto dell'evento custom generato dal bottom component e poi gestirlo:

```
<app-header (featureSelected)="onNavigate($event)"> </app-header>

loadedFeature:string = 'recipes'
onNavigate(feature:string) {
  console.log(feature)
  this.loadedFeature = feature
}
```

In corrispondenza dell'evento **featureSelected** andiamo a richiamare il metodo onNavigate() definito all'interno del file di classe

Al metodo onNavigate() viene passato come parametro **\$event** che è quello visto nella sezione di gestione degli eventi, cioè una parola chiave che indica ad angular di **propagare i parametri alla procedura di gestione dell'evento** (parametri iniettati dal bottom component in fase di generazione dell'evento).

**loadedFeature** è una stringa che alla fine conterrà il testo del pulsante cliccato all'interno del bottom component ("recipes" oppure "shopping-list")

**Note:**

Notare che l'evento può essere propagato al componente superiore (top component) ma non ad eventuali altri componenti ancora superiori. Se il parametro deve essere propagato anche a componenti successivi, il componente intermedio deve a sua volta generare un evento custom da inviare al componente extra-superiore e così via.

Questa tecnica degli eventi custom va bene per passare parametri fra componenti direttamente interconnessi. In caso di variabili che devono essere condivise fra molti componenti costituenti l'applicazione conviene utilizzare i **services** che consentono di condividere i dati con tutti i componenti.

**Parametri multipli**

Al metodo `emit()` in realtà è possibile passare anche un intero json per cui, se serve passare più parametri, si può passare un intero json creando più campi per ogni informazione da passare:

```
@Output() featureSelected = new EventEmitter<any>();

onSelect(feature: string) {
  let json = {"feature":feature, "cod":97}
  this.featureSelected.emit(json)
}
```

**Ascolto di un evento da codice: il metodo subscribe()**

Quando un Bottom Component genera un evento tramite il metodo `emit()`, il Top Component può mettersi all'ascolto all'interno dell'html come indicato negli esempi precedenti, ma può mettersi in ascolto anche da Type Script.

A tal fine deve disporre di un puntatore al Bottom Component ottenibile tramite il decoratore `ViewChild`:

```
@ViewChild(ChildComponent, {static:false}) _child : ChildComponent;
```

A questo punto il Top Component può mettersi in ascolto dell'evento nel modo seguente:

```
ngOnInit(): void {
  _child.featureSelected.subscribe(
    (feature:string) => {
      console.log(feature)
      this.loadedFeature = feature
    }
  )
}
```

In sostanza il componente corrente (nell'esempio `app-component`), si mette in ascolto permanente dell'evento `featureSelected` generato da un altro componente puntato da `_child`.

Tutte le volte che si verifica l'evento `featureSelected` verrà automaticamente eseguito il metodo `subscribe` di gestione dell'evento al quale viene iniettato il parametro `feature:string` che, nel caso dell'esempio, viene visualizzato e salvato all'interno di una variabile locale.

Per ottenere il puntatore al Bottom Component si potrebbe anche far uso dei Template Reference :

```
<component1 #disabler></component1>
<component2 #disabler></component2>
<component3 #disabler></component3>

@ViewChildren('disabler') mycomponents: QueryList<any>;
this.mycomponents.forEach((element)=>{element.disable();})
```

## Utilizzo di bootstrap come libreria interna al progetto

Per poter utilizzare bootstrap all'interno del progetto sarebbe ovviamente sufficiente specificare il link CDN all'inizio di tutti i `file.html` oppure scaricare bootstrap in locale ed inserire in tutte le pagine un link all'installazione locale.

In alternativa, in Angular, è possibile linkare la libreria direttamente al progetto rendendola così fruibile in tutte le varie pagine. Lanciando dalla cartella principale di lavoro il comando,

```
npm install bootstrap@3 // 3.4.1
```

bootstrap viene installato nella sottocartella `node_modules` con relativo aggiornamento del `package.json`

A questo punto occorre però referenziare bootstrap all'interno dell'applicazione, andando a 'registrare' bootstrap all'interno del file `angular.json` che contiene tutte le configurazioni relative al progetto.

All'interno del file `angular.json` cercare **styles**

In questa sezione sono elencati tutte le librerie css che devono essere utilizzate dall'applicazione:

```
"styles": [      // riga 26
  "node_modules/bootstrap/dist/css/bootstrap.min.css",
  "src/styles.css"
],
```

Aggiungere il riferimento a bootstrap prima del riferimento al file `src/styles.css`, in modo che gli stili personali siano prevalenti rispetto a quelli definiti all'interno di bootstrap.

La stessa cosa si potrebbe fare alla riga 86 relativa alla sezione di test, ma non è strettamente necessario.

Se si usasse anche la componente js di bootstrap, occorrerebbe fare la stessa cosa anche per il js:

```
"scripts": [
  "node_modules/bootstrap/dist/js/bootstrap.js"
],
```

Ogni volta che si fanno delle modifiche al file `angular.json` occorre **riavviare** il server.

Per vedere l'effettivo funzionamento di bootstrap si può inserire un button con una classe bootstrap:

```
<button class="btn btn-primary"> invia </button>
```

## ngForm and Template Reference

Per poter utilizzare **ngForm** ed i **Template Reference** occorre eseguire l'import di **FormsModule** esattamente come per **ngModel**

**ngForm** è una Direttiva esportata da **FormsModule** che viene automaticamente aggiunta a tutti i tag **<form>** nei modelli Angular una volta importato il modulo. La direttiva **ngForm** crea un'istanza nascosta **FormGroup** e la associa al tag **<form>** per consentirti di lavorare con il modulo. Ad esempio per controllare lo stato di convalida.

Si può ottenere un riferimento alla classe **ngForm** esportandolo in una **variabile di modello** (template reference) ed utilizzando la direttiva **ngForm** come valore della variabile:

```
<form #myform="ngForm" (ngSubmit)="register(myform)" >
```

All'intero della form dovrà esserci un pulsante di submit in corrispondenza del quale verrà eseguita la procedura `register()` alla quale viene passato il puntatore alla `ngForm`.

Tramite quato puntatore, all'interno della procedura `register()`, si potranno utilizzare lei seguenti proprietà / metodi:

**myform.value**: returns the aggregated form url-encoded value of all the fields used in your `<form>` tag,  
**myform.valid**: returns a boolean value indicating if the form is valid or not,  
**myform.touched**: returns a boolean value indicating if the user has entered value at least in one field,  
**myform.submitted**: returns a boolean value indicating if the form was submitted.

### Esempio:

```
<form #myform="ngForm" (ngSubmit)="register(myform)" >
  <label for="register-email">Email</label>
  <input name="email" type="email" placeholder="Email" ngModel>

  <label for="register-password">Password</label>
  <input name="password" type="password" placeholder="Password" ngModel >

  <input type="submit" value="Register">
</form>
```

```
export class RegisterComponent implements OnInit {
  constructor() { }
  ngOnInit() {}

  register(form) {
    console.log(form.value);
    console.log(form.touched);
    console.log(form.submitted);
    console.log(form.controls['email'].value);
    console.log(form.controls['password'].value);
    form.resetForm(); // ripulisce tutti i campi
  }
}
```

In alternativa, senza utilizzare **ngForm**, è comunque possibile assegnare un **Template Reference** ai singoli elementi della pagina html, elementi che diventeranno poi accessibili nella classe associata facendo uso del decoratore **@ViewChild()**:

```
<input type="text" id="name" name="name" #nameInput>
<input type="number" id="amount" name="amount" #amountInput>

@Output() ingredientAdded = new EventEmitter<IngredientModel>();
@ViewChild('nameInput') nameInputRef: ElementRef
@ViewChild('amountInput') amountInputRef: ElementRef;
addIngredient() {
  const ingName: string = this.nameInputRef.nativeElement.value;
  const ingAmount: number = this.amountInputRef.nativeElement.value;
  const newIngredient: IngredientModel
    = new IngredientModel(ingName, ingAmount);
  this.ingredientAdded.emit(newIngredient);
}
```

## Direttive personalizzate

Oltre a nuovi Componenti è possibile anche creare nuove Direttive personalizzate ognuna legata ad una **singola** CSS property oppure ad una **singola** classe.

Una direttiva personalizzata può essere creata manualmente come semplice classe oppure può essere autogenerata tramite CLI fermando il server ed utilizzando il seguente comando:

```
ng generate --help
ng generate directive shared/highlight --skip-tests
```

- Viene generata una cartella **shared** (facoltativa) con dentro il file **highlight.directive.ts**
- La Direttiva si chiamerà **HighlightDirective**

Alla classe **HighlightDirective** viene applicato automaticamente il decoratore `@Directive()` il quale associa un selettore per l'utilizzo della Direttiva all'interno dei vari componenti

```
@Directive({
  selector: '[appHighlight]'
})
```

La classe viene anche automaticamente registrata fra le **declarations: []** nel file **app.module.ts**

```
declarations: [
  ... ,
  HighlightDirective,
],
```

La sintassi di utilizzo è sostanzialmente la stessa, indipendentemente dal fatto che si tratti di una Direttiva di Attributo oppure di una Direttiva di Classe.

### Direttive di Attributo:

```
export class HighlightDirective implements OnInit {
  @Input('appHighlight') hoverColor: any = 'Cyan';
  @Input() defaultColor: any = 'LightCyan';
  @HostBinding('style.backgroundColor') backgroundColor: any =
    this.defaultColor;

  ngOnInit() { // this rappresenta la classe corrente
    this.backgroundColor = this.defaultColor;
  }
  @HostListener('mouseenter') evidenzia(event: Event) {
    this.backgroundColor = this.hoverColor;
  }
  @HostListener('mouseleave') rilascia(event: Event) {
    this.backgroundColor = this.defaultColor;
  }
}
```

Il primo parametro **hoverColor** è un parametro di tipo `Input('appHighlight')` ed indica dove memorizzare il valore che il chiamante eventualmente assegnerà alla Direttiva **appHighlight** al momento dell'istanza:

```
<a [appHighlight]="Yellow">
```

Questo parametro, avente tra le parentesi della `Input()` il nome stesso della direttiva funge in pratica da 'costruttore' della Direttiva. Dopo di che il chiamante potrà assegnargli o meno un valore.



Nell'esempio in questione questa proprietà, denominata `hoverColor`, indica il colore di sfondo che dovrà assumere il componente ospite al momento del mouseover.

- Se il selettore `appHighlight` viene invocato senza l'assegnazione di un valore, la Property `hoverColor` assumerà automaticamente il valore di default 'Cyan'
- Diversamente assumerà il valore passato come parametro

Il secondo parametro `defaultColor` indica una property aggiuntiva di tipo `Input()` che potrà essere assegnata sempre dal componente ospite. Questa proprietà, denominata `defaultColor`, indica il default del colore di sfondo che dovrà avere il componente ospite.

- Se la property `defaultColor` non viene assegnata in fase di istanza, assumerà automaticamente il valore di default 'LightCyan'
- Diversamente assumerà il valore passato come parametro

Il terzo parametro `backgroundColor` è una property che sarà 'linkata' al `backgroundColor` dell'elemento ospite. Viene inizializzata al valore di `defaultColor`. In corrispondenza del `mouseover` assume il valore `hoverColor` e poi ritorna a `defaultColor` in corrispondenza del `mouseout`

Il decoratore `@HostBinding()` fa sì il valore della Property `backgroundColor` venga 'collegato' alla proprietà di Stile '`style.backgroundColor`' del componente ospite.

In questo modo la nostra Direttiva viene collegata ad un Attributo del componente ospite, da cui il nome di Direttiva di Attributo.

#### Metodi:

- all'avvio (all'interno di `ngOnInit`) si inizializza il `backgroundColor` del componente ospite (linkato alla proprietà `backgroundColor` della classe corrente) al valore `defaultColor`
- la procedura `evidenzia`, in corrispondenza del `mouseenter` sul componente ospite, assegna allo sfondo del componente il valore corrente del campo `hoverColor`.  
Il parametro event non viene utilizzato per cui può anche essere omissso.
- la procedura `rilascia`, in corrispondenza del `mouseleave` sul componente ospite, ri-assegna allo sfondo del componente il valore `defaultColor`. Il parametro event può anche essere omissso

#### Esempi di utilizzo completo

```
<a [appHighlight]='Yellow' [defaultColor]='LightYellow' >
<a [appHighlight]='Yellow' [defaultColor] >
<a [appHighlight] [defaultColor]='LightYellow' >
<a [appHighlight] [defaultColor] >
```

#### Note:

1. Le Direttive Personalizzate e le eventuali proprietà aggiuntive vengono di solito richiamate **facendo uso delle parentesi quadre** per far capire ad Angular che la stringa di destra contiene (di solito) una variabile (nota successiva n. 4)
2. Se la Direttiva Personalizzata NON presenta un costruttore, allora il componente ospite **dovrà** richiamare la Direttiva Personalizzata **senza fare uso delle parentesi quadre**.
3. Il componente ospite **potrà** richiamare la Direttiva Personalizzata **senza fare uso delle parentesi quadre** anche in caso di presenza del costruttore, nel qual caso però `hoverColor` risulterà `undefined`. Per evitare questo problema si può modificare `evidenzia()` nel modo seguente  
`this.backgroundColor = this.hoverColor ? this.hoverColor : "cyan";`

4. A destra dell'uguale è prevista sempre una stringa contenente di solito il nome della variabile associata.
- Se a destra dell'uguale non si usa una variabile ma un valore diretto, se questo valore diretto è una stringa, tale stringa deve essere scritta utilizzando ulteriori apici interni.  
Se invece il valore diretto fosse un numero oppure true/false, allora deve essere scritto direttamente all'interno della stringa principale.
  - Se a destra dell'uguale si utilizza un valore diretto (stringa, numero o booleano), si potrebbero omettere le parentesi quadre sul nome della Direttiva e scrivere il valore di destra in forma diretta senza avvolgerlo all'interno di una sottostringa. Sorge però un problema: se l'elemento è di tipo **number** oppure **boolean** occorre necessariamente utilizzare le parentesi quadre perché:  
`appHighlight = 7` produrrebbe syntax error (a destra ci vuole sempre una stringa)  
`appHighlight = "7"` assegnerebbe la stringa "7" invece del number 7  
 Per cui, alla fine, l'abitudine è quella di usare sempre le quadre intorno al nome della direttiva
5. La property `hoverColor` associata al costruttore non può essere richiamata esplicitamente scrivendo ad esempio
- ```
<a [appHighlight] [hoverColor]='Yellow' [defaultColor]='LightYellow'
```
- Per rendere corretta la riga precedente occorre definire una seconda proprietà aggiuntiva **hoverColor** ed assegnare alla proprietà del costruttore un nome ausiliario tipo **aus** oppure cancellare completamente il primo parametro di tipo Input (cioè il 'costruttore').

### Direttive di Classe:

```
export class DropdownDirective {
  @HostBinding('class.open') isOpen: boolean = false;
  @HostListener('click') toggleOpen(event: Event) {
    this.isOpen = !this.isOpen;
  }
}
```

Le Direttive di Classe di solito non hanno parametri in `Input()` da parte del componente ospite.

**Non avendo costruttore dovranno essere scritte all'interno del componente ospite senza parentesi quadre.**

- All'interno della Direttiva viene definita una unica variabile booleana **isOpen** inizialmente false 'collegata' alla classe `'open'` del componente ospite.  
'open' è una classe **bootstrap** che quando viene applicata ad un drop down menù lo apre, mentre quando non viene applicata il menù si chiude
- In corrispondenza del click sul componente ospite la variabile inverte il proprio valore. Il parametro `eventData` può anche essere omissso. Per cui quando **isOpen** è true la classe **open** viene applicata al componente ospite (che viene aperto) e viceversa.

Questa Direttiva Personalizzata può essere utilizzata per aprire/chiudere un qualsiasi drop down menù.

Esempio di utilizzo all'interno di un componente ospite:

```
<ul class="nav navbar-nav navbar-right">
  <li class="dropdown" appDropdown>
    <a style="cursor:pointer;" class="dropdown-toggle" role="button">
      Manage
      <span class="caret"></span>
    </a>
    <ul class="dropdown-menu">
      <li><a style="cursor:pointer;">Save Data</a></li>
      <li><a style="cursor:pointer;">Fetch Data</a></li>
    </ul>
  </li>
</ul>
```

## I servizi

Un servizio è una semplice classe che implementa delle funzionalità e/o gestisce dei dati che possono essere condivisi fra tutti i vari componenti che costituiscono l'applicazione.

Un servizio, oltre che essere utilizzato dai componenti, può, a sua volta, sfruttare le funzionalità fornite da altri servizi. Nelle prime versioni il servizio era indicato con il termine *provider*

In generale è buona regola sollevare i componenti dagli incarichi di logica “business” concentrandola invece all'interno dei Servizi. Un servizio è solitamente rappresentato da una classe indipendente dalle View che viene definita per svolgere un compito ben preciso nel rispetto del principio di singola responsabilità: ogni servizio si occupa di svolgere un unico e ben preciso compito.

Un tipico esempio di servizio è quello per l'invio centralizzato di richieste Ajax ad un REST server.

Ogni servizio è costituito da un unico file `.ts` come per le Directive.

Il service può essere creato manualmente come semplice classe oppure può essere autogenerato tramite CLI fermando il server ed utilizzando il seguente comando:

```
ng generate service services/recipe --skip-tests
```

In fase di creazione Angular aggiunge automaticamente al servizio una desinenza **Service** e soprattutto un decoratore **@Injectable()** che rende il servizio injectable, cioè rende l'intera classe iniettabile al costruttore di un qualsiasi altro componente o servizio. Questo meccanismo per cui i servizi possono iniettarsi uno dentro l'altro è detto Dependency Injection. Molto comodo per utilizzare i servizi senza dover censire ogni singolo servizio all'interno del file `app.module.ts`.

Per cui il comando precedente creerà un servizio:

- che si chiamerà **RecipeService** (desinenza Service)
- Avrà applicato un decoratore **@Injectable()** così strutturato :

```
@Injectable({  
  providedIn: 'root' })
```

- Viene posizionato all'interno di un file denominato `recipe.service.ts`
- Non dovrà essere registrato all'interno nel file `app.module.ts`

Tutti i componenti a cui verrà iniettato il servizio avranno accesso condiviso a Proprietà e Metodi del servizio medesimo.

### Servizio di gestione delle richieste Ajax

Andiamo a creare un servizio denominato **DataStorageService** che si occupi di gestire tutte le chiamate http in modo centralizzato.

```
ng generate service shared/data-storage --skip-tests
```

Questo servizio si occuperà di gestire tutte le chiamate http verso un API server remoto.

All'inizio del servizio corrente occorre importare manualmente

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

Il modulo angular che si occupa di gestire le chiamate http è il modulo **HttpClientModule** che deve essere importato manualmente all'interno degli imports di **app.module.ts**.

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

imports: [
  BrowserModule,
  FormsModule,
  AppRoutingModule,
  HttpClientModule
],
```

Andiamo quindi ad impostare il servizio **DataService** a cui andiamo ad iniettare nel costruttore il componente **HttpClient** che, attenzione, non è **HttpClientModule** inserito nel file app.module.ts, ma una sua sottoclasse. L'import iniziale invece è il medesimo

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

@Injectable({
  providedIn: 'root'
})
export class DataService {
  private REST_API_SERVER = "http://localhost:3000/";
  constructor(private httpClient: HttpClient) { }
  public getRequest(endpoint: string) {
    return this.httpClient.get(this.REST_API_SERVER + endpoint);
  }
}
```

- Creiamo all'inizio una Property **REST\_API\_SERVICE** (che è una costante ma, come per tutte le Property, occorre omettere **const** / **let**)
- All'interno del costruttore viene creata al volo una istanza della classe **HttpClient**
- Aggiungiamo quindi un metodo **getRequest()** che è un wrapper del metodo **this.httpClient.get()** che si aspetta due parametri: una URL obbligatoria e poi un json di opzioni facoltativo che può contenere diversi parametri aggiuntivi come ad esempio il tipo dei dati restituiti (che per default è un json)

### Invio delle richieste ed utilizzo dell'oggetto Observable

Tutti i metodi dell'oggetto **HttpClient** restituiscono al chiamante un oggetto **Observable** che è un oggetto analogo all'oggetto **Promise** di java script e che consente al chiamante di mettersi in attesa asincrona della risposta. L'oggetto **Observable** rende disponibile un unico metodo di evento denominato **.subscribe()** il quale verrà richiamato sia in corrispondenza dell'arrivo dei dati sia in corrispondenza di un errore.

In pratica il componente asincrono **Observable** si mette in ascolto della ricezione della risposta e, quando arriva, richiama il metodo di evento **'subscribe()'** iniettandogli il json ricevuto dal server

Il metodo **subscribe()** si aspetta come parametri due funzioni di callback, una success ed un error

- Alla funzione di success vengono iniettati i dati restituiti dal server in formato json
- Alla funzione di error viene iniettato un oggetto **err** che è anche lui un json contenente varie informazioni relative all'errore.

---

## Esempio completo di utilizzo del servizio

---

Si supponga di voler inviare una richiesta GET in corrispondenza del caricamento di un componente:

```
constructor(public dataStorageService: DataStorageService) {  
    this.dataStorageService.getRequest('recipes').subscribe(  
        (data: any[]) => {  
            this.recipes = data;  
            this.selectedRecipe = this.recipes[0];  
        },  
        (error: any) => {  
            console.log(error);  
        }  
    )  
}
```

### Note

- Quando si inietta un servizio all'interno di un costruttore occorre utilizzare **public** e non **private** perché diversamente i membri del servizio iniettato sarebbero visibili soltanto nella classe e non all'esterno, in particolare non sarebbero visibili all'interno del file **html**.
- **Sui Service l'evento onInit() non è significativo e non viene generato.** Eventuali inizializzazioni devono essere eseguite all'interno del costruttore.
- Volendo utilizzare una variabile intermedia di tipo **Observable** occorre importare l'oggetto  

```
import { Observable } from 'rxjs';  
let ris:Observable<any>;  
ris = this.dataStorageService.getRequest('recipes')  
ris.subscribe(.....)
```
- E' ovviamente possibile anche definire degli Observable tipizzati con un particolare formato JSON

---

## Conversione di un Observable in Promise

---

L'oggetto **Observable** di TypeScript può facilmente essere convertito in una **Promise** JavaScript facendo semplicemente uso del metodo **.toPromise()**

Il codice precedente potrebbe essere riscritto nel modo seguente:

```
let promise = this.dataStorageService.getRequest('recipes').toPromise();  
promise.then( (data: RecipeModel[]) => {  
    console.log(data);  
    this.recipes = data;  
    this.selectedRecipe = this.recipes[0];  
})  
promise.catch( (error: any) => {  
    console.log(error);  
})
```

---

## Utilizzo delle Arrow Function all'interno delle callback

---

Sia nel caso delle **Promise** che nel caso dell'oggetto **Observable**, per la scrittura delle funzioni di callback si utilizzano solitamente le arrow function perché, lavorando sempre all'interno di metodi di classe, **nel caso delle arrow function il this mantiene anche all'interno delle funzioni di callback il contesto a monte rispetto alle funzioni asincrone in cui viene inserito, cioè continua sempre a puntare alla classe corrente.**

Viceversa se si utilizzano per le callback le normali funzioni "tradizionali", al loro interno il **this** rappresenterebbe l'oggetto Promise / Observable all'interno del quale il this è stato utilizzato.

## Il controllo dell'applicazione tramite i Services

Invece di inviare le http request all'onLoad del componente, si può creare un ulteriore servizio intermedio che, mediante il **DataStorageService** precedente, si occupi di inviare le richieste http e di prendersi in carico tutta la gestione delle ricette (aggiornamento di una ricetta, cancellazione, inserimento di una nuova ricetta), snellendo al massimo il codice dei componenti che si occuperanno soltanto più della visualizzazione dei dati

Attraverso l'utilizzo dei servizi si cerca di portare tutta la logica "business" all'interno dei servizi piuttosto che distribuirla all'interno dei vari componenti. Concetto di **Controller** tipico del paradigma **MVC**:

**Model** -> Struttura dei dati (es ricette e ingredienti)  
**View** Rappresentazione dei dati attraverso i **Componenti**  
**Controller** Gestione centralizzata della "business logic" attraverso i **Servizi**

Per ogni componente di livello più alto dell'applicazione (le cosiddette **features**) si va a creare un relativo servizio che ne gestisce l'intera logica di funzionamento.

Si può ad esempio creare un servizio per la gestione delle ricette.

```
ng generate service recipes/recipe --skip-tests
```

All'interno del costruttore iniettiamo il **DataStorageService** precedente

```
constructor(private dataStorageService: DataStorageService) { }
```

Definiamo quindi un metodo **getRecpies()** che esegue il codice precedente di invio della richiesta di tutte le ricette appoggiandosi al sottostante **dataStorageService.getRequest()**

### Completamento del DataStorageService con un interfaccia CRUD

```
public sendGetRequest(resource: string) {  
    return this.httpClient.get(this.REST_API_SERVER + endpoint);  
}  
public sendDeleteRequest(resource: string) {  
    return this.httpClient.put(this.REST_API_SERVER + endpoint);  
}  
public sendPostRequest(resource: string, body:any) {  
    return this.httpClient.post(this.REST_API_SERVER + endpoint, body);  
}  
public sendPatchRequest(resource: string, body:any) {  
    return this.httpClient.patch(this.REST_API_SERVER + endpoint, body);  
}
```

L'ID del record da aggiornare / eliminare viene passato come **parte della risorsa**: es recipe/2

### Completamento dei Business Services

A questo punto i servizi di livello più alto possono gestire un metodo per ciascuna delle operazioni CRUD  
Ad esempio:

```
updateRecipe(id:number, body:any){  
    this.dataStorageService.sendPatchRequest('recipes/'+id, body).subscribe(  
        (data: any[]) => { },  
        (error: any) => { }  
    )  
}
```

va ad aggiornare la ricetta avente l'id indicato con i campi contenuti all'interno del parametro body.

## Salvataggio dei dati all'interno del componente

Aniché salvare i dati all'interno del servizio è preferibile salvarli all'interno del componente, lasciando all'interno del servizio SOLTANTO i metodi di richiesta dati al server.

**Ogni componente al momento del `ngOnInit()` richiede i dati aggiornati al server e se li salva localmente. In questo modo disporrà sempre di dati aggiornati, evitando di leggere dati 'vecchi' all'interno della cache del servizio, e si migliora decisamente anche la gestione del refresh.**

A tal fine il componente DEVE gestire il `subscribe()` del servizio; oppure meglio ancora potrebbe passare la funzione di elaborazione come callback:

**Esempio:** Si vogliono richiedere i dettagli di un film avente un certo `id`:

**// MAIN**

```
// Passo come secondo parametro una function anonima a cui verrà iniettato data
this.service.getFilmDetails(this.id, (data) => {
  this.currentFilm = data;
});
```

**// SERVICE**

```
getFilmDetails( id: number,
  // definizione del 2° parametro del tipo: void function callback(data) (*)
  callback: (data: any) => void,
  // definizione del 3° parametro facoltativo void function error(err) (**)
  error = (err) => { console.log(err)} )
{ // corpo della funzione
  this.dataStorage.getRequest(URL + "/film/" + id).subscribe(callback,error)
}
```

**(\*)** La seconda riga `callback: (data: any) => void` specifica il tipo di callback ricevuta. Se non si specifica il tipo della callback, si genera un errore sulla **`subscribe()`** successiva

**(\*\*)** La terza riga `error = (err) => { console.log(err)}` specifica un parametro facoltativo `error` che è un puntatore a funzione che di solito NON viene passato dal chiamante. In caso di omissione del parametro viene assegnata la funzione scritta sopra che dovrà ricevere un parametro `err` e che, come corpo della funzione, eseguirà semplicemente un `console.log(err)`

Dopo di che il metodo `getFilmDetails()` richiama il metodo di livello inferiore

`this.dataStorage.getRequest()` passandogli la URL ed i parametri di filtro necessari e mettendosi poi in ascolto della risposta tramite il metodo `subscribe()`. Al metodo `subscribe()` vengono passati come parametri la funzione di **callback** ricevuta dal chiamante e la funzione **error** scritta sopra come parametro facoltativo. Quando il servizio di basso livello riceverà i dati dal server richiamerà la funzione di callback ricevuta dal chiamante che andrà a salvare i dati nella variabile `this.currentFilm`

**Conclusione:** In genere i servizi non fanno storage e richiedono sempre i dati al server.

In questo modo non solo è garantito che i componenti abbiano sempre dati aggiornati, ma viene notevolmente migliorato e semplificato il refresh delle pagine. Ogni pagina di dettaglio per default, **mantiene i propri parametri di routing**; se poi, sulla base di questi parametri, richiede i dati necessari al server, tutto funzionerà perfettamente.

### Nota:

Aniché utilizzare l'evento `ngOnInit()` il componente può richiamare il servizio all'interno dell'evento `ionViewDidEnter()` che viene richiamato tutte le volte che il componente viene visualizzato. Esiste anche un evento `ionViewWillLeave()` richiamato quando il componente perde il focus.



## Routing delle pagine

Quando un utente accede ad un'applicazione Web o a un sito Web, si intende per "routing" il meccanismo che consente la navigazione all'interno dei vari componenti che costituiscono l'applicazione.

- Lavorando single page, se una pagina gestisce contenuti multipli, tutte le ricerche punteranno alla stessa pagina senza alcuna differenziazione.
- Sarebbe invece preferibile avere una URL che ci dica se stiamo visualizzando le ricette oppure la shopping-list. Organizzando infatti la app attraverso un meccanismo di routing si migliora l'indicizzazione da parte dei motori di ricerca. In questo modo i vari componenti verranno visti come "pagine" ciascuna con una propria URL fittizia e, di conseguenza, i motori di ricerca potranno puntare alla pagina corretta. Inoltre in tal modo si potrebbe accedere direttamente ai dettagli della ricetta dalla barra di navigazione scrivendo `/recipes/2`

Per semplificare la gestione del routing, Angular fornisce un apposito modulo chiamato **Router** che è una alternativa più strutturata rispetto all'utilizzo di `ngIf` e gestisce il routing tramite la definizione di apposite routes. Le varie routes devono essere definite manualmente all'interno di un apposito "modulo utente" che di solito viene denominato **AppRoutingModule** e che viene definito all'interno del file `src/app/app-routing.module.ts`

Quando si crea una nuova applicazione, se la si crea utilizzando il flag `--routing`

```
ng new my-app --routing
```

automaticamente Angular crea il modulo **AppRoutingModule** all'interno del file `app-routing.module.ts`

Una volta creata l'applicazione è possibile creare il modulo **AppRoutingModule** tramite il seguente comando:

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

dove:

`--flat` puts the file in `src/app` instead of its own folder.

`--module=app` tells the CLI to register it in the `imports` array of the `AppModule`.

La seconda opzione aggiunge `AppRoutingModule` fra gli imports del file `app.module.ts`

```
imports: [  
  BrowserModule,  
  FormsModule,  
  AppRoutingModule  
],
```

### Struttura di **AppRoutingModule**

Inizialmente viene eseguito l'import delle seguenti due librerie:

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

Occorre quindi definire una costante di tipo `Routes []`, denominata solitamente `appRoutes` che è un array in cui occorre definire tutte le varie `routes` necessarie all'applicazione.

Nel progetto delle Ricette si inizia con il definire due `Routes` di primo livello, che devono essere scritte **senza lo slash** e che verranno automaticamente concatenate al path di dominio della pagina corrente:

- **recipes** in corrispondenza della quale viene visualizzato il componente `RecipesComponent`
- **shopping-list** in corrispondenza della quale viene visualizzata la `ShoppingList`

```
const appRoutes: Routes = [  
  {  
    path: '',  
    redirectTo: '/recipes',  
    pathMatch: 'full'  
  },  
  {  
    path: 'recipes',          // senza slash !  
    component: RecipesComponent,  
  },  
  {  
    path: 'shopping-list',    // senza slash !  
    component: ShoppingListComponent  
  }  
];  
  
@NgModule({  
  imports: [RouterModule.forRoot(appRoutes)],  
  exports: [RouterModule],  
  // providers: []  
})  
export class AppRoutingModule { }
```

- Per ogni route l'unico parametro obbligatorio è **path** che definisce il nome della route. Le **routes** di primo livello devono essere scritte come routes relative (senza slash) che però il chiamante dovrà "passare" all'oggetto **Router** antepoendo lo slash davanti. Ad es **/recipes**
- Le routes, in linea di massima, possono essere di due tipi:
  - **routes** di caricamento di un **component**,
  - **routes** che eseguono un **redirect**.

Il path iniziale (vuoto) è un path di redirect ed indica che cosa bisogna visualizzare quando l'utente accede al sito senza specificare nessuna risorsa. Nel nostro caso vogliamo eseguire un redirect al path `/recipes`. L'opzione **pathMatch: 'full'** è obbligatoria e sta ad indicare che il match deve essere eseguito per intero, cioè deve andare a buon fine SOLO quando l'utente inserisce veramente stringa vuota. Diversamente il match andrebbe a buon fine anche quando l'utente inserisce un qualunque altro path indicato nelle route successive.

- Il decoratore **@NgModule** riceve un oggetto di inizializzazione che prevede la presenza dei campi **imports** ed **exports**. `RouterModule` è il modulo importato all'inizio del quale si invoca il metodo `.forRoot ()` passandogli come parametro il vettore delle routes. Queste due righe sono "fisse" e servono sostanzialmente ad inizializzare il router.
- `providers: []` viene creato in automatico con il comando **new app**. Può essere utilizzato per definire delle variabili globali oppure, nel caso di Ionic, per definire i plugin di cordova

### **Navigazione delle routes da html e da codice: <router-outlet>**

Nella sezione principale della APP (tipicamente all'interno di **app-component**) si può inserire una tipica barra di navigazione che contiene i link alle varie "pagine" del sito (es ricette e shopping-list).

Sempre all'interno di **app-component** si può inserire un nuovo componente detto **<router-outlet>** che fungerà da visualizzatore per i vari componenti indicati all'interno della costante **appRoutes**.

Ogni volta che dalla barra di navigazione oppure da codice si imposta una nuova route, angular provvederà a visualizzare all'interno di **<router-outlet>** il componente associato a quella route

Di seguito si riporta una tipica semplicissima struttura d un **app-component**:

```
<app-header></app-header>
<router-outlet> </router-outlet>
```

Per indicare al router quale route visualizzare all'interno del componente **<router-outlet>** ci sono due possibilità:

- 1) Direttamente nell'html facendo uso di un semplice collegamento ipertestuale **<a>** ed impostando all'interno dell'attributo **routerLink** la route da caricare. Per cui all'interno della **nav-bar** si possono inserire vari collegamenti ipertestuali scritti nel modo seguente che consentiranno di caricare all'interno di **<router-outlet>** il componente associato alla route:

L'attributo **routerLink** agisce in modo simile all'attributo **<a href>** e si aspetta come parametro il path da caricare:

```
<li><a routerLink="/"> Home </a></li> // ricarica la pagina
<li><a routerLink="/recipes">Recipes</a></li>
<li><a [routerLink]="['/recipes']">Recipes</a></li>
<li><a [routerLink]="['/shopping-list']">Shopping List</a></li>
```

Il valore della url può anche essere passato come vettore di stringhe che saranno poi automaticamente concatenate per comporre la url. Es `"['/recipes', '/2']"`  
Questo facilita l'utilizzo delle variabili all'interno della url.

Se la stringa da passare è una sola le quadre possono essere omesse.

Se vogliamo utilizzare delle variabili per indicare il path, è **obbligatorio l'uso delle parentesi quadre a sinistra dell'uguale (ovviamente) ma anche a destra**

- 2) L'alternativa consiste nel gestire l'evento **click()** su un qualunque tag e poi caricare la pagina da Type Script utilizzando il metodo **router.navigate()**. Atal fine occorre però iniettare al costruttore il componente Injectable **Router**:

```
<li> <a (click)="caricaRicette()"> Recipes </a> </li>

constructor(private router:Router) { }
caricaRicette() {
    this.router.navigate(['/recipes'])
}
```

Anche **router.navigate()** come **routerLink** si aspetta come parametro un vettore di stringhe. Il componente **router** viene dichiarato **private** in quanto non è necessario che sia visibile all'interno dell'html.

Una terza possibilità consiste nello scrivere direttamente nella barra di navigazione

```
http://localhost:4200/recipes
http://localhost:4200/shopping-list
```

## L'attributo routerLinkActive

La Direttiva **routerLink** dispone di un ulteriore comodissimo attributo **routerLinkActive** che, applicato a tutte le voci di un menù, consente di applicare automaticamente una certa classe alla voce attualmente selezionata. Si può utilizzare questo attributo per aggiungere la classe bootstrap **active** alla voce di menù attualmente selezionata:

```
<li routerLinkActive="active"><a routerLink="/recipes">Recipes</a></li>
<li routerLinkActive="active"><a routerLink="/shopping-list">ShList</a></li>
```

Attenzione che l'attributo **routerLinkActive** funziona soltanto per quei link definiti via html attraverso la Direttiva **routerLink** e NON per le sezioni eseguite da codice mediante il metodo **router.navigate()**.

## Routes parametriche (dinamiche)

Si supponga ora di voler visualizzare sempre all'interno del medesimo **<router-outlet>** di **app.component** i dettagli di una specifica ricetta, ad esempio la ricetta avente id=2.

Questa pagina potrà essere richiamata dalla barra di navigazione scrivendo

```
http://localhost:4200/2
```

Aggiungiamo nel vettore delle **routes** la seguente route sempre di primo livello.

Notare i **due punti** che indicano appunto che si tratta di una route parametrica, a cui potrà essere passata qualsiasi informazione, numerica o alfanumerica.

```
{
  path: ':id',
  component: DettagliRicettaComponent
}
```

Questa route potrà essere richiamata nei seguenti modi:

```
<li><a routerLink = "2"> Dettagli ricetta 2</a></li>
<li><a [routerLink]="[id]">Dettagli ricetta</a></li>

caricaDettagli() {
  this.router.navigate(['2'])
}
```

**Notare che**, nel momento in cui si definisce una route parametrica, qualunque richiesta che NON soddisfi alle altre routes presenti nel vettore (nel nostro caso **/recipes** e **/shopping-list**) verrà automaticamente intercettata dalla route parametrica:

```
http://localhost:4200/2
http://localhost:4200/xyz
http://localhost:4200/aaaaaaa
```

**Per cui ad ogni livello dell'albero delle routes ci può essere una unica route parametrica che deve essere scritta per ultima fra le routes di pari livello** e che intercetterà tutte le richieste non specificate nelle routes precedenti.

## Lettura dei parametri

Il componente `DettagliRicetta`, richiamato in corrispondenza del richiamo di un link, avrà necessità di accedere all'id della ricetta richiesta in modo da poter richiedere le informazioni al server e procedere alla visualizzazione.

Per poter accedere ai parametri passati ad una route parametrica, occorre innanzitutto iniettare al costruttore del componente `DettagliRicetta` l'oggetto `ActivatedRoute` contenente la route attualmente selezionata:

```
constructor( private route:ActivatedRoute) {}
```

L'oggetto `ActivatedRoute` genera un evento `params` ogni volta che viene attivata una qualsiasi route associata al componente corrente (parametrica o non parametrica).

In caso di route parametrica (che inizia con i due punti) alla funzione di gestione dell'evento viene iniettato un json NON VUOTO contenente il parametro di attivazione della route, in questo caso il parametro `id`. Per cui, in questo caso, il json iniettato alla funzione di gestione dell'evento, sarà:

```
{"id":2}
```

Il metodo `.subscribe()` definisce, all'interno della classe corrente, un listener di evento che intercetterà tutti gli eventi `params` generati ogni volta che una route porta al caricamento del componente attuale. All'interno dell'evento si può accedere all'ID e richiedere al server i dettagli della ricetta selezionata

**L'associazione** tra l'evento `subscribe()` e la procedura di gestione dell'evento dovrà essere eseguita in fase di avvio del componente, cioè all'interno del metodo `onInit()`

```
ngOnInit(): void {
  this.route.params.subscribe(
    (params: Params) => {
      if(params.id) {
        console.log(params['id'])
        this.recipeService.getRecipe(params['id']);
      }
    }
  )
}
```

Oppure molto più semplicemente si può utilizzare il seguente metodo sincrono:

```
let detailsId =this.route.snapshot.paramMap.get("id");
```

**Notare però** che questo metodo sincrono non è del tutto equivalente alla soluzione precedente.

Utilizzando questa soluzione, nel caso di routes parametriche, nel momento in cui si passa da una route parametrica all'altra (ad esempio da `/user/1` a `/user/2`) **l'evento `onInit()` sul componente associato NON viene generato** (o meglio viene generato solo in corrispondenza della prima visualizzazione). Viceversa la soluzione con il `subscribe()` forza una specie di 'reset' sul componente associato in modo che l'evento `onInit()` venga generato in corrispondenza di ogni variazione della route parametrica.

## Child routes

Riprendendo l'esempio precedente, il fatto di posizionare la route parametrica al primo livello non è la soluzione migliore. Sarebbe meglio impostare la route parametrica come sotto-route di `recipes`, in modo che l'utente possa richiamare i dettagli di una ricetta nel modo seguente:

```
http://localhost:4200/recipes/2
```

A tal fine si possono aggiungere nel modulo di routing, all'interno del path relativo alle ricette, le seguenti child-routes figlie della route primaria **recipes**

```
{
  path: 'recipes',
  component: RecipesComponent,
  children: [
    { path: '', component: RecipeStartComponent },
    { path: 'new', component: RecipeNewComponent },
    { path: ':id', component: RecipeDetailComponent },
    { path: ':id/edit', component: RecipeEditComponent }
  ]
},
```

- Il secondo path, in corrispondenza della route **recipe/new**, carica un componente per l'inserimento di una nuova ricetta
- Il terzo path, in corrispondenza della route **recipe/2** carica il componente **recipe-detail** per la visualizzazione dei dettagli della ricetta indicata
- Il quarto path caricherà un componente per l'editazione della ricetta corrente, precaricando tutti i valori correnti.
- Il primo path interviene nel momento in cui l'utente richiede **/recipes** senza specificare nessun id, nel qual caso caricherà nell'area di destra un apposito componente praticamente vuoto contenente il solo messaggio

```
<p>Please select a Recipe!</p>
```

### Richiamo delle child-routes

Le child routes relative ai vari sottocomponenti potranno essere richiamate da html oppure da TypeScript nel solito modo:

```
<a [routerLink]="[recipe.id]" routerLinkActive="active">
```

Notare che, a differenza delle routes di primo livello, **le child routes passate a routerLink NON devono iniziare con lo slash**, proprio perchè sono routes relative che verranno automaticamente concatenate alla route corrente. Il componente contenente questo link è un componente che sarà visualizzato soltanto in corrispondenza della route **/recipes**.

Pertanto, in corrispondenza del click, Angular provvederà a concatenare il valore di **routerLink** alla route corrente, producendo come risultato ad esempio **/recipes/2**

Da Type Script tramite utilizzo del metodo **navigate()**

Lavorando da Type Script il concatenamento precedente non è automatico ma occorre specificare un secondo parametro che indichi a quale route deve essere concatenato il "path relativo" contenuto nel primo parametro. Il valore del secondo parametro sarà praticamente sempre **this.route**

```
constructor(private router: Router, private route: ActivatedRoute) { }
onNewRecipe() {
  this.router.navigate(['new'], { relativeTo: this.route })
}
```

In caso di click sui dettagli della ricetta corrente **/recipes/2** andiamo a concatenare **"edit"** per il caricamento del componente preposto all'editazione del record corrente

```
this.router.navigate(['edit'], { relativeTo: this.route })
```

## Criterio di visualizzazione delle routes

Quando viene impostata una route mediante click sul un tag `<a>` oppure mediante il metodo `router.navigate()`, Angular in prima battuta verifica se si tratta di una **route assoluta** (`/recipes`) oppure di una **child-route** (`/recipes/new`)

- Se si tratta di una **route assoluta**, il componente associato alla **route** viene caricato all'interno del `<router-outlet>` di `app.component`, il quale deve sempre avere al suo interno un `<router-outlet>`. Se non ci fosse, semplicemente la route non viene 'servita'
- Se si tratta invece di una **child-route**, Angular guarda alla **Parent-Route** (`/recipes`). Se la **Parent-Route** (`/recipes`) ha un componente assegnato, questo componente dovrebbe sempre avere un `<router-outlet>`. In pratica, la situazione 'corretta' è che :  
**ogni componente che presenti delle child-routes deve sempre esporre un `<router-outlet>` in cui verranno visualizzati i componenti 'puntati' dalle child-routes.**

Per cui lo scenario precedente può essere completato assegnando a `RecipesComponent` un `<router-outlet>` in cui visualizzare **TUTTI** i sotto componenti associati alle varie **child-routes**.

Quindi, all'interno di `RecipesComponent`, si potranno definire due sezioni:

- Una sezione di sinistra contenente la lista delle ricette più alcuni pulsanti, ad esempio per l'inserimento di una nuova ricetta
- Una sezione di destra contenente il `<router-outlet>` in cui verranno visualizzati i dettagli della ricetta selezionata e, ad esempio, una form per l'inserimento di una nuova ricetta.

```
<div class="row">
  <div class="col-md-5">
    <app-recipe-list></app-recipe-list>
    <button> Add </button> <button> Edit </button>
  </div>
  <div class="col-md-7">
    <router-outlet> </router-outlet>
  </div>
</div>
```

## Casi Particolari

- Se la **Parent-Route** (`/recipes`) ha un componente assegnato, e questo componente non espone un `<router-outlet>`, allora la route non viene servita
- Se la **Parent-Route** (`/recipes`) **NON** ha un componente assegnato (e dunque non può esporre un `<router-outlet>`), allora Angular provvede a scorrere l'albero dei componenti verso l'alto finché non individua un selettore `<router-outlet>`, fino ad arrivare eventualmente ad `app.component`. Nel caso arrivi fino ad `app.component` sostituisce però completamente la Main Page attualmente visualizzata. Non va comunque bene.

A tal proposito, al di là del funzionamento, sorge spontanea una domanda: come si fa a dichiarare una route senza assegnargli un componente? Si usa un piccolo 'trucco' riportato di seguito:

<https://medium.com/dev-genius/the-art-of-nested-router-outlets-in-angular-dafb38245a30>

```
export const routes: Routes = [
  { path: '', redirectTo: 'mainpage', pathMatch: 'full' },
  { path: 'mainpage',
    children: [
      { path: '', component: MainpageComponent },
      { path: 'first', component: Tab1Component },
      { path: 'second', component: Tab2Component }
    ]
  }
];
```

`mainPage` non ha un componente associato ma viene visualizzata comunque grazie alla childroute vuota



## Lazy Loading Modules

Per impostazione predefinita, i moduli Angular vengono caricati nel momento in cui l'app viene caricata, indipendentemente dal fatto che siano immediatamente necessari oppure no. Per app di grandi dimensioni con molti percorsi, è previsto un modello di progettazione che consente di caricare gli NgModule solo quando necessari. Il **Lazy Loading** (letteralmente caricamento lento) aiuta a mantenere le dimensioni dei bundle iniziali più piccole, il che a sua volta aiuta a ridurre i tempi di caricamento.

Per poter utilizzare il Lazy Loading di un certo componente (es. `admin`) occorre creare non un semplice componente ma un NgModule utilizzando il seguente comando:

```
ng generate module admin --route admin --module app.module
```

comando che genera un componente `admin` completo comprensivo di due files aggiuntivi:

```
admin.module.ts
admin-routing.module.ts
```

Il 1° file contiene l'import di tutti i moduli necessari all'esecuzione del componente. Eventuali moduli aggiuntivi dovranno essere inseriti in questo file e saranno caricati al momento del caricamento di questo modulo. In questo modo il file globale `app.module.ts`, contenente i files da caricare all'avvio, risulta molto più leggero e di rapida esecuzione.

La sezione `declarations: [ ]` dichiara ad Angular l'esistenza del componente.

Il 2° file contiene le informazioni di routing relative al file corrente ed è quasi sempre espresso nel seguente formato:

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

Il comando precedente utilizzato senza parametri,

```
ng generate module admin
```

crea semplicemente un modulo all'interno del quale si possono poi aggiungere manualmente uno o più componenti.

L'opzione `--module app.module` è obbligatoria in presenza dell'opzione `--route`

L'opzione `--route admin` fa sì che venga creato l'intero componente anche e venga anche aggiornato il file di routing generale `app-routing.module` con l'aggiunta della seguente route:

```
{ path: 'admin',
  loadChildren: () => import('./admin/admin.module')
    .then(mod => mod.AdminModule)
}
```

Cioè, in corrispondenza della richiesta `/admin`, anziché specificare il Component da caricare, come specificato nelle routes iniziali, si utilizza il metodo `loadChildren()` che consente di caricare il componente soltanto nel momento in cui l'utente invocherà il path `/admin`.

Il metodo `loadChildren()` si aspetta come parametro una function priva di parametri all'interno della quale viene invocata la funzione typescript `import` che restituisce una Promise. In corrispondenza del termine del caricamento viene generato l'evento `then()` la cui funzione di callback deve restituire il **modulo** da caricare. All'interno del modulo c'è il nome del componente.

Senza aver scritto nessuna riga di codice, se si apre l'applicazione e si digita nella barra di navigazione `/admin` il router provvederà automaticamente a caricare all'interno di `<router-outlet>` la pagina html del componente `admin`.

## Pubblicazione di un progetto Angular su Firebase

Firebase è un servizio di Google Cloud che consente di ospitare diverse tipologie di applicazioni web.

Link:

<https://firebase.google.com/>

Breve Guida:

<https://stackoverflow.com/questions/42573987/how-to-host-angular-2-website>

Prima di pubblicare su Firebase occorre eseguire un build di produzione del progetto angular:

```
ng build --prod
```

il quale crea una cartella di output **dist** con all'interno tutti i files javascript necessari per la pubblicazione

L'opzione **--prod** serve a settare a true il flag angular di "production configuration" che ripulisce la app da tutte le informazioni di debug e la rende pubblicabile su un server.

Verificare all'interno del file `angular.json` che il campo `outputPath` sia impostato a `dist` oppure `dist/projectName` e verificarne la correttezza. In caso contrario può essere modificato manualmente. Il TypeScript viene tradotto in javascript, tutto il routing viene sostituito con dei tag `<a href=".....">`

## Creazione del progetto su Firebase

Per **registrarsi** a Firebase occorre disporre di un **Account Google** ed eseguire i seguenti passi:

- In corrispondenza del primo accesso vengono visualizzati eventuali progetti già impostati all'interno della Google Developer Console
- **Creare un nuovo progetto Firebase.** Assegnare un nome (es **progetti-angular** o **ricette-mediterranee**) utilizzando la tipica notazione URL, cioè sono consentiti solo caratteri minuscoli, numeri e trattino per una lunghezza max di 30 caratteri. I progetti sono dei semplici contenitori per le APP. In uno stesso progetto possono essere aggiunte più APP
- Al progetto viene automaticamente assegnato un **SITE\_ID** univoco che sarà poi utilizzato come URL di accesso al sito (a cui viene automaticamente accodato il suffisso **.web.app**). Il **SITE\_ID** può comunque essere modificato in un secondo tempo dal menù SETTINGS/GENERALE

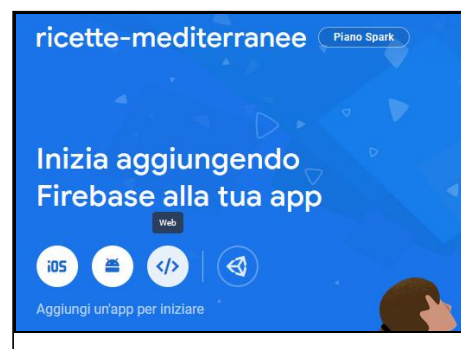
Se il nome del progetto è univoco sull'intero dominio **web.app**, come **SITE\_ID** viene assegnato lo stesso nome del progetto. Se non è univoco vengono concatenati dei numeri in modo da renderlo univoco. Anche in questo caso è utilizzabile la tipica notazione URL, cioè sono consentiti solo caratteri minuscoli, numeri e trattino per una lunghezza max di 30 caratteri.

- Eventualmente (**ma non è il caso**) abilitare il servizio di **Google Analytics** che abilita un insieme di funzionalità di test e reporting relativi al progetto. In caso di abilitazione di Google Analytics occorre configurare la località del progetto (Italia).

### **CREA PROGETTO**

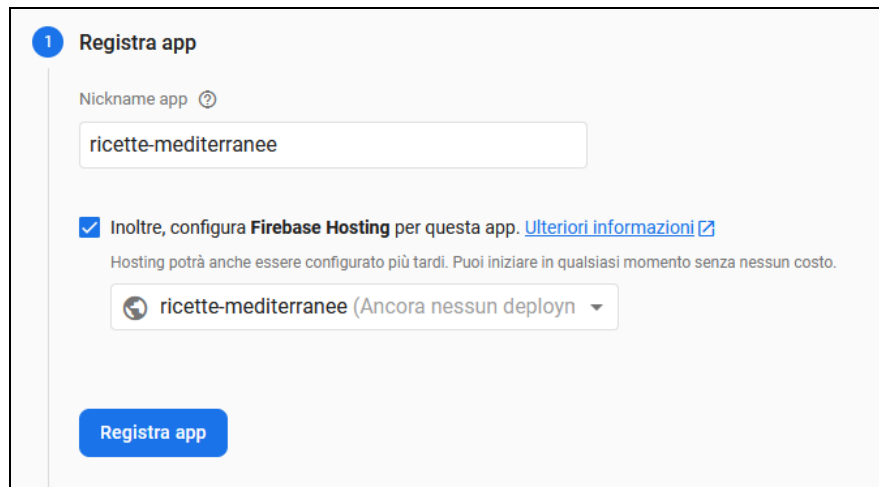
Aggiungere quindi una **WEB APP** al progetto, scegliendo tra:

- IOS
- Android
- **Web**
- Unity



Il **nickname** è semplicemente un nome per identificare la app all'interno della firebase console.

Abilitare il **Firestore**



Dopo aver registrato la APP vengono proposti alcuni tag da incollare in coda a **index.html** prima del **tag </body>**. La loro presenza non sembra indispensabile, però meglio seguire le indicazioni

```
<!-- The core Firebase JS SDK is always required and must be listed first -->
<script src="/__/firebase/8.3.2/firebase-app.js"></script>

<!-- TODO: Add SDKs for Firebase products that you want to use
https://firebase.google.com/docs/web/setup#available-libraries -->
<script src="/__/firebase/8.3.2/firebase-analytics.js"></script>

<!-- Initialize Firebase -->
<script src="/__/firebase/init.js"></script>
```

## Publicazione (da linea di comando)

- 1) Installare Il Firebase CLI per comunicare a linea di comando con il server

```
npm install -g firebase-tools
```

- 2) **firebase login**

Da fare **SOLO in corrispondenza del primo accesso**. Viene aperta una finestra del browser in cui scegliere l'account google da utilizzare per l'accesso a Firebase. Al termine viene visualizzato un messaggio di successo e la finestra del browser può essere chiusa. Sul terminale si può vedere un messaggio che segnala che il login è stato effettuato con successo.

- 3) **firebase init**

Viene richiesto quale servizio di firebase intendiamo utilizzare. Selezionare **Hosting: Configure and deploy Firebase Hosting site.**

Selezionare quindi il **SITE\_ID** del progetto Firebase creato nei punti precedenti. Questo **SITE\_ID** viene memorizzato all'interno del file **.firebaserc** e successivamente non verrà più richiesto. In caso di necessità potrà eventualmente essere modificato a mano.

Viene quindi richiesta la **build output directory** (per default viene proposto **public**)

Digitare **dist** oppure **dist/projectName** a seconda del contenuto del file **angular.json**

Le ultime tre domande richiedono :

- se l'applicazione è una single page con reidrect automatico di tutte le URL to index.html (Y)
- se salvare su gitHub tutti i build e deploy (N)
- se Firebase deve sovrascrivere il file index.html (N)

Al termine vengono automaticamente creati i files firebase.json e .firebaseerc.

#### 4) firebase deploy

Al termine il progetto sarà raggiungibile all'indirizzo

`https://ricette-mediterranee.web.app/`

Se su firebase si hanno più progetti attivi:

`firebase projects:list` consente di vedere la lista dei progetti attivi

`firebase use ricette-mediterranee` consente di 'switchare' sul progetto indicato

### Progetti Multisites

---

Nel momento in cui si dovesse aggiungere al progetto una seconda APP, anche per la seconda APP occorre definire un **SITE\_ID** univoco. Dopo di che in fase di deploy occorre indicare su quale APP deve essere scaricato il progetto. A tale scopo si rimanda alla documentazione ufficiale:

`https://firebase.google.com/docs/hosting/multisites`

### Dashboard

---

Dalla dashboard si possono visionare i vari accessi, deploy, etc.

### CORS Problems

---

Nel momento in cui la app deve accedere ai dati di un web server (ad esempio heroku) sul server si manifestano problemi CORS relativi a richieste provenienti da domini esterni. Per cui, lato server, occorre gestire con attenzione le impostazioni CORS:

## Creazione di una apk angular - cordova

La creazione di una apk cordova non interferisce con la creazione di una build per Firebase in quanto ognuno crea i propri file all'interno di cartelle differenti (**dist** per firebase e **www** per cordova)

Dalla cartella di lavoro lanciare il comando **setenv** per l'impostazione delle variabili d'ambiente.

Nel caso di ionic occorre aggiungere la seguente variabile d'ambiente:

`set _JAVA_OPTIONS=-Xmx512M`

che serviva per impostare correttamente le dimensioni della memoria.

Con java 1.8 non serve più anzi sembrerebbe creare problemi.

1) installare `npm install -g cordova-res`

2) Eseguire una build di produzione tramite il comando

```
ng build --prod --base-href . --output-path ../www/
```

#### Significato delle opzioni

L'opzione **--prod** serve a settare a true il flag angular di "production configuration" che rende l'apk installabile sul device. Senza questa opzione l'apk non potrà essere installata sul device. Anche con l'opzione --prod si tratta però sempre di una apk di debug non adatta per il rilascio su uno store.

L'abbinamento delle opzioni **--prod --release** serve invece per creare una apk non di tipo "debug" ma di tipo "release" rilasciabile sugli stores. Attenzione però che una apk di tipo "release" deve essere controfirmata con le apposite chiavi, altrimenti non solo non sarà pubblicabile sugli stores, ma non sarà nemmeno installabile sui propri device.

L'opzione **--base-href .** should be used to set the base reference to "." as absolute paths are not well handled by Cordova

L'opzione **--output-path** indica il percorso dove salvare la build di produzione. La documentazione raccomanda di salvare la build di produzione in una cartella diversa rispetto alla cartella di lavoro di angular in modo da non creare interferenza fra le librerie di angular e quelle di cordova.

3) Creare una nuova applicazione cordova e sostituire la cartella **www** con la cartella appena creata al punto precedente.

4) Aggiungere in coda ad index.html (prima del </body>) il link a cordova.js.

```
<script type="text/javascript" src="cordova.js"></script>
```

5) Lanciare quindi i soliti comandi cordova:

```
cordova platform add android
cordova build
```

L'apk viene salvata nel percorso `\platforms\android\app\build\outputs\apk\debug`

### **Esecuzione diretta della app**

---

Come per tutte le app cordova è possibile eseguire la app direttamente senza doverla installare su un device. A tal fine occorre :

- Installare **npm install -g native-run**
- Aprire un emulatore oppure collegare un device
- Lanciare uno dei seguenti comandi:

```
cordova run android // build + run su emulatore o device
cordova run android --no-build // solo run (installa soltanto)
```

### **Debug dell'applicazione**

---

Aprire **chrome://inspect**

automaticamente visualizza l'eventuale dispositivo collegato (e/o l'emulatore) e, all'interno del dispositivo, l'elenco delle app attualmente in esecuzione.

Selezionare la app desiderata e cliccare su **inspect**,

Cliccare sul pulsante **INSPECT**

Esattamente come per tutte le applicazioni cordova

## Cenni su Ionic

ionic è un tool di sviluppo di tipo SPA che può utilizzare Angular/React/Vue e che mette a disposizione una serie di componenti ottimizzati per mobile e basati principalmente su Angular.

```
https://ionicframework.com/  
https://ionicframework.com/docs/api
```

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

```
ionic version // versione installata
```

### Creazione di un progetto blank

Seguire **Ionic Getting Started**

```
ionic start ese01 blank
```

L'ultima voce (blank) indica il template da utilizzare. Se non viene specificata verrà richiesta negli step successivi. I template disponibili, a marzo 2021, sono i seguenti:

```
> tabs           | A starting project with a simple tabbed interface  
  sidemenu       | A starting project with a side menu with navigation in the content area  
  blank          | A blank starter project  
  list           | A starting project with a list  
  my-first-app   | An example application that builds a camera with gallery  
  conference     | A kitchen-sink application that shows off all Ionic has to offer
```

Alla domanda “**Integrate your new app with Capacitor to target native iOS and Android**” rispondere NO. Capacitor è un nuovo framework alternativo a Cordova per il build dell'applicazione,

Alla domanda “**Install the free Ionic AppFlow SDK...**” rispondere NO.

### Esecuzione:

L'alberatura che viene creata è la tipica alberatura di una applicazione Angular.

Entrare nella cartella di lavoro e digitare il comando

```
ionic serve --open
```

che avvia un server sulla porta 8100 al quale ci si può connettere direttamente da browser :  
localhost:8100

### Home Page

L'applicazione crea una pagina di apertura quasi vuota denominata **Home**, costituita dai 3 +2 files tipici del lazy loading (cioè con l'aggiunta del file di **Modulo** e del file di **Routing**)

La pagina di apertura **app.component.html** contiene al suo interno un unico tag

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

**app-routing.module.ts**

contiene una unica route con di seguito il solito redirect in caso di path vuoto

```
const routes: Routes = [
  {
    path: 'home',
    loadChildren:
      () => import('./home/home.module').then( m => m.HomePageModule)
  },
  {
    path: '',
    redirectTo: 'home',
    pathMatch: 'full'
  }
];
```

Cioè in corrispondenza dei path `/` oppure `/home` l'applicazione provvederà a caricare tramite Lazy Loading il modulo `home.module.ts` il quale manderà in esecuzione il componente `home.page.html`

**home-routing.module.ts**

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

**Contenuto della pagina home.html**

Le pagine ionic sono di solito articolate tramite un `<ion-header>` seguito da uno `<ion-content>` suddivisi a livello grafico da una riga orizzontale che rappresenta il bordo inferiore della `<ion-header>`

```
<ion-header [translucent]="true">
  <ion-toolbar>
    <ion-title>
      Blank
    </ion-title>
  </ion-toolbar>
</ion-header>

<ion-content [fullscreen]="true">
  <ion-header collapse="condense">
    <ion-toolbar>
      <ion-title size="large">Blank</ion-title>
    </ion-toolbar>
  </ion-header>

  <div id="container">
    <strong>Ready to create an app?</strong>
  </div>
</ion-content>
```

**Il componente `<ion-header>`**

contiene una `<ion-toolbar>` contenente soltanto un titolo, cioè Blank



Gli attributi `[translucent]="true"` e `[fullscreen]="true"` sono **Direttive Personalizzate** collegate fra loro e raccomandate dalla documentazione.

Notare come siano scritte in forma indiretta con l'uso delle parentesi quadre al fine per poter inserire il valore booleano `true` all'interno di una stringa. La parte destra di una direttiva personalizzata deve sempre essere una stringa. Omettendo le parentesi quadre il `true` verrebbe quindi interpretato come stringa e potrebbe essere causa di problemi.

### Il componente `<ion-content>`

---

Contiene un secondo `<ion-header>` identico al precedente però questa volta nascosto (`collapse="condense"`). Questo doppio `<ion-header>` dovrebbe servire nel caso di IOS per fornire una versione expanded del menù. Spiegazione dalla documentazione ufficiale:

*In the example above, notice there are two **ion-header** elements. The first **ion-header** represents the "collapsed" state of your collapsible header, and the second **ion-header** represents the "expanded" state of your collapsible header. Notice that the second **ion-header** must have `collapse="condense"` and must exist within **ion-content**. Additionally, in order to get the large title styling, **ion-title** must have `size="large"`.*

Nel caso di Android questo `<ion-header>` può tranquillamente essere eliminato.

Segue quindi il contenuto vero e proprio della pagina che potrà essere impostato in base alle proprie esigenze

### Creazione di una nuova pagina

---

In Ionic non si creano semplici componenti ma pagine. Una pagina è un componente con in più due files

- `component.module.ts`
- `component-routing.module.ts`

`ionic generate page details --skip-tests`

oppure

`ionic g`

Selezionare **Page**

Digitare il nome della pagina: **Details**

All'interno della cartella app verrà creata una nuova sottocartella **details** contenente tutti i 5+1 files relativi alla nuova pagina. Viene automaticamente aggiornato anche il file **app-routing.modules.ts**

```
const routes: Routes = [  
  { path: 'home', loadChildren: './home/home.module#HomePageModule' },  
  { path: '', redirectTo: 'home', pathMatch: 'full' },  
  { path: 'details', loadChildren: './details/details.module#DetailsPageModule' }  
];
```

La nuova pagina può essere raggiunta digitando `/details` direttamente nella barra di navigazione

## Creazione di un nuovo servizio

```
ionic generate service shared/data-storage --skip-tests
```

All'inizio del servizio corrente occorre importare manualmente

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

Occorre poi importare manualmente il modulo **HttpClientModule** all'interno degli imports di **app.module.ts**.

```
import { HttpClientModule } from '@angular/common/http';
imports: [
  BrowserModule,
  FormsModule,
  AppRoutingModule,
  HttpClientModule
],
```

## I componenti predefiniti

La documentazione fornisce una descrizione molto completa dei vari componenti al seguente indirizzo:

<https://ionicframework.com/docs/api>

### ion-button

E' un componente inline che può essere personalizzato nei seguenti modi:

#### color

```
<ion-button color="primary">Primary</ion-button>
<ion-button color="secondary">Secondary</ion-button>
<ion-button color="tertiary">Tertiary</ion-button>
<ion-button color="success">Success</ion-button>
<ion-button color="warning">Warning</ion-button>
<ion-button color="danger">Danger</ion-button>
<ion-button color="light">Light</ion-button>
<ion-button color="medium">Medium</ion-button>
<ion-button color="dark">Dark</ion-button>
```



#### size

**small**   **default**   **large**

il large aumenta le dimensioni del carattere ed il padding

#### expand

**block**   **full**

Entrambi provocano una visualizzazione del pulsante sull'intera larghezza del genitore

Il primo mantiene il leggero bordino e gli spigoli arrotondati

Il secondo elimina gli spigoli arrotondati ed i bordi destro e sinistro

**fill****clear outline solid**

PRIMARY

SECONDARY

TERTIARY

Il valore **solid** consente di mantenere il colore di sfondo quando si usa il pulsante all'interno di una toolbar, che per default diventerebbe trasparente

**Altri attributi**

**href** – eventuale link da aprire in corrispondenza del click. Il link verrà aperto nella scheda corrente oppure in una nuova scheda impostando `target="_blank"`

**disabled** - boolean

**strong** – visualizza il testo in grassetto

**type** – può essere `button` / `reset` / `submit`

**download** – consente di aprire la finestra di download della risorsa anziché navigare verso la risorsa, esattamente come l'attributo `download` del tag `<a>` di html. L'eventuale valore assegnato all'attributo `download` rappresenta il valore da preimpostare come "filename" all'interno della finestra di download.

**routerDirection** – può essere **back** / **forward** / **root**. Abbinato a `routerLink` applica un effetto grafico di scroll al caricamento della nuova pagina. Può assumere i seguenti valori:

**forward** [default] - la nuova pagina viene caricata dal basso verso l'alto

**back** - la nuova pagina viene caricata dall'alto verso il basso

**root** - la nuova pagina sostituisce di getto la vecchia pagina senza nessuna animazione

**ion-input**

E' il tipico textbox. Riconosce tutti i principali attributi html:

**type** che può essere "text", "password", "file", "email", "number", "search", "tel", "url"

**value**, **placeholder**, **disabled**, **readonly**, **maxlength** (number of chars), **required**

**color** colore del testo

**pattern** regex di controllo del contenuto

**clearInput** Aggiunge una X a destra che consente di cancellare il contenuto del textbox

Per default il componente `<ion-input>` è di tipo block (occupa tutta la riga) ma può essere trasformato tramite in **display: inline-block**; con eventuale indicazione della width

Per default non presenta nessun bordo perché in genere viene utilizzato all'interno di un componente `<ion-item>` che presenta lui il bordino.

Il bordo può comunque essere aggiunto da CSS: `ion-input { border: 1px solid black; }`

Riconosce gli eventi classici **ionInput** e **ionChange**. Se occorre intercettare il codice del tasto

premuto, si possono utilizzare gli eventi javascript **keyup** e **keydown** che iniettano al parametro **event** il codice ascii del tasto premuto:

```
(keydown)="onKeydown($event) "
```

```
onKeydown(e) { alert(e.keyCode) }
```

E' anche possibile intercettare la pressione di un singolo tasto (es il tasto enter) nel modo seguente:

```
(keydown.enter)="onKeydown($event) "
```

Che equivale sostanzialmente a **ionChange**

## ion-label

E' equivalente del tag `<span>` di tipo inline a cui possono essere applicati tutti i vari stili CSS.

La classe `class="ion-text-wrap"` consente l'a capo automatico del testo. Senza questa classe il testo viene visualizzato tutto su una riga. Al suo interno riconosce i normali tag html.

**Color** colore del testo

**position** per default la label occupa una larghezza pari al suo contenuto. Il valore `position="fixed"` fa sì che la label assuma una larghezza fissa in modo da avere allineamento fra le varie righe.

Sono possibili anche i valori `floating` e `stacked`

## ion-item

E' uno dei componenti più importanti di ionic. E' un contenitore di tipo `display:block` che può raggruppare diversi tipi di componenti e visualizza un bordo inferiore (per suddividerlo dal prossimo item).

E' possibile tramite CSS modificare spessore e colore del bordo (vedasi documentazione)

Un item è **clickable** quando utilizza le proprietà `href` oppure `button`. Nel qual caso presenta una grafica leggermente diversa

Property:

**button** = true/false L'item assume l'aspetto grafico di un pulsante e diventa graficamente cliccabile

**href / target** indica la risorsa da caricare e l'eventuale nuova scheda

**download** il link indicato da href potrà essere scaricato invece che fungere da navigazione.

**detail** = true/false Se true aggiunge una freccia a destra per far capire che, cliccando si apre un'altra finestra di dettaglio. Di solito è true sugli item clickable.

**detailIcon** icona da utilizzare nel caso di `detail=true`.

**lines** = full/inset/none Modalità di visualizzazione del bottom-border

**disabled** disabilita l'intero item.

**slot** = start/end Applicato ad un elemento interno all'item, lo posiziona all'inizio o alla fine dell'item, ma al di fuori dell'item stesso (cioè fuori dalla sottolineatura). **E' simile a `float:left/right`**

Non ha eventi. Per la navigazione si può usare `routerLink`

## ion-list

Consente di visualizzare una lista di elementi. Contiene di solito un elenco di `<ion-item>`. Prima dell'inizio degli `<ion-item>` può esserci una `<ion-list-header>` contenente un titolo.

La `<ion-list>` non presenta proprietà particolarmente rilevanti.

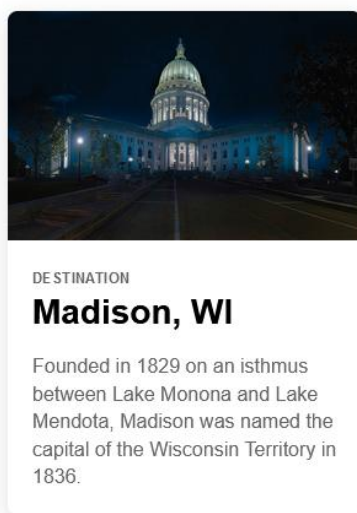
Gli `<ion-item>` possono essere raggruppati all'interno di uno o più `<ion-item-group>` oppure all'interno di una `<ion-card>` basica che li racchiude all'interno di una cornice.

## ion-card

Anch'esso molto utilizzato. Usato da solo funge tipicamente da contenitore per gli `<ion-item>`. Può però contenere al suo interno diversi sottocomponenti:

Esempio di utilizzo:

```
<ion-card>
  
  <ion-card-header>
    <ion-card-subtitle>Destination</ion-card-subtitle>
    <ion-card-title>Madison, WI</ion-card-title>
  </ion-card-header>
  <ion-card-content>
    lorem ipsum dolor sit amet .....
  </ion-card-content>
</ion-card>
```



## ion-checkbox

I checkbox dispongono delle seguenti properties:

**checked, color, disabled, name, value.** **indeterminate**=true/false (aspetto grafico 'indeterminato'),

Dispongono inoltre di tre eventi:

- **ionChange** richiamato quando il contenuto del checkbox cambia di valore.
- **ionFocus** e **ionBlur** richiamati rispettivamente quando il checkbox acquisisce o perde il focus

## ion-radio

Sono alternativi solo se inseriti all'interno di un `<ion-radio-group>`

In corrispondenza del **click** viene generato l'evento (`ionChange`) sul `<ion-radio-group>`

Come parametro alla funzione di gestione dell'evento si può passare **\$event** il cui campo target rappresenta il puntatore all'elemento che ha scatenato l'evento:

```
(ionChange)="getStudents($event) "
```

Il gestore di evento può accedere al value della voce selezionata tramite `event.target.value`

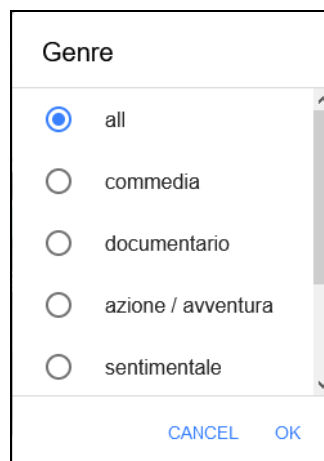
```
getStudents(e) {  
  let radioValue = e.target.value
```

I singoli radiobutton dispongono di pochissime properties (**color, disabled, name, value**) e di soli due eventi: **ionFocus** e **ionBlur** richiamati rispettivamente quando il radiobutton acquisisce o perde il focus

## ion-select

E' un listbox che però, per la scelta della voce, apre una apposita finestra di scelta:

```
<ion-item>  
  <ion-label>Genre</ion-label>  
  <ion-select value="-1" (ionChange)="onGenreChange($event)">  
    <ion-select-option value="0">all</ion-select-option>  
    <ion-select-option *ngFor="let genere of genereList" [value]="genere.id">  
      {{genere.nome}}  
    </ion-select-option>  
  </ion-select>  
</ion-item>
```



L'evento ion-change viene generato in corrispondenza del click sul pulsante ok.

## ion-text

Contenitore di tipo display:inline che può contenere **solo testo** (non può contenere controlli).  
Consente una unica cosa: impostare il colore del testo per l'intero contenuto:

```
<ion-text color="secondary" class="ion-hide">
  <h1>H1: The quick brown fox jumps over the lazy dog</h1>
</ion-text>
```

## ion-icon

Sono disponibili moltissime icone accesibili semplicemente attraverso il **name**  
L'attributo **size** consente di definire le dimensioni: **small** / **default** / **large**

## ion-img

simile al tag html **img**. Però dispone di un tag di chiusura e soprattutto esegue un lazy load dell'immagine solo nel momento in cui questa viene visualizzata all'interno del viewport (a schermo)  
A differenza del tag **<img>** è di tipo display:block;

## ion-avatar

Avatars are **circular components** that usually wrap an image or icon.

```
<ion-avatar slot="start">
  <img [src]="student.picture">
</ion-avatar>
```

## ion-thumbnail

Thumbnail are **square components** that usually wrap an image or icon.

```
<ion-thumbnail slot="end">
  <img [src]="student.picture">
</ion-thumbnail>
```

E' possibile definire la dimensione del thumbnail mediante la proprietà `--size: 180px;`  
Se la dimensione non viene espressa si allarga fino a riempire l'area parent.

## Store delle immagini in locale

Eventuali immagini salvate all'interno dell'applicazione devono essere salvate all'interno della cartella **src/assets**, eventualmente all'interno di una sottocartella **img**

Per accedere mediante un tag **<img>** si può utilizzare la seguente sintassi:

```
<img *ngFor="let item of vect" [src]='assets/img/'+item.filename">
```

## Ion-grid

All'interno dei contenitori come **<ion-card>** è possibile utilizzare il componente **<ion-grid>** che applica il tipico grid system di bootstrap:



```
<ion-grid>
  <ion-row>
    <ion-col size="2"></ion-col>
    <ion-col size="8">
      .....
    </ion-col>
    <ion-col size="5"></ion-col>
```

## Componenti per il routing

Il routing è sostanzialmente lo stesso di Angular

All'interno della toolbar di una pagina di dettagli si può aggiungere un pulsante di ritorno:

```
<ion-button slot="start" [routerLink]="['/home']" routerDirection="back">
  <ion-icon slot="start" name="arrow-back"> </ion-icon>
</ion-button>
```

Oppure in alternativa si potrebbe utilizzare un pulsante predefinito `<ion-back-button>` eventualmente inserito all'interno di un contenitore `<ion-buttons>`

```
<ion-buttons slot="start">
  <ion-back-button defaultHref="/home"> </ion-back-button>
</ion-buttons>
```

Il componente `<ion-back-button>` non dispone della proprietà `href`, ma soltanto della proprietà `defaultHref` che indica la risorsa da caricare in assenza di `history`. Quando esiste una `history` di navigazione viene ricaricata l'ultima pagina indipendentemente da `defaultHref`. Deve però comunque essere presente, altrimenti il pulsante non viene visualizzato.

In entrambi i casi al ritorno sulla pagina principale non viene più richiamato il costruttore ma nemmeno `ngOnInit()`. Mi ritrovo cioè i dati precedenti come se avessi cliccato sul pulsante back del browser.

## CSS custom properties

La documentazione ufficiale riporta, in coda ad ogni ionic-tag, un elenco di "CSS custom properties" disponibili per quell'elemento. Si tratta di proprietà CSS ridefinite con una sintassi più snella ed utilizzabili all'interno del file `css` di pagina per modificare più rapidamente l'aspetto grafico dell'elemento.

Non ci sono però solo proprietà `css` ridefinite ma anche nuove proprietà difficilmente realizzabili tramite i soli `CSS`. Ad esempio nel caso dei **radiobuttons** è disponibile una proprietà `--color` che consente di definire il colore da assegnare al cerchio del radiobutton, cosa tutt'altro che semplice da fare con i soli `css`. La proprietà `--color-checked` rappresenta invece il colore da assegnare al radiobutton selezionato.

```
ion-radio {
  --color: #006;
  --color-checked: #66F; }
```

All'interno del file `.css` si possono utilizzare preferibilmente queste custom properties ma, in caso di necessità, anche i `CSS` tradizionali.

## CSS utilities

Sono **classi** che possono essere aggiunte ad un qualunque elemento per controllare l'aspetto grafico. Ad esempio `class="ion-hide"` (equivalente a `display:none`).

## Componenti Aggiuntivi

Oltre ai componenti 'tradizionali' utilizzabili all'interno del DOM, è possibile utilizzare molti altri componenti aggiuntivi ([//docs/v3/api/components/](https://docs/v3/api/components/)) come ad esempio il componente `AlertController` che consente di realizzare diverse finestre di dialogo graficamente più curate rispetto alla alert tradizionale. Questi componenti devono essere:

- importati all'inizio del file type script
- iniettati al costruttore della classe esattamente come per i servizi.  
Il componente verrà istanziato al momento dell'injection

### Visualizzazione di una finestra di dialogo: `AlertController`

La versione **basic** è molto simile al componente predefinito **ion-toast**

```
import { AlertController } from '@ionic/angular';  
constructor(public alertController: AlertController) { }
```

Aggiungiamo quindi alla classe il seguente metodo (**basic alert**)

```
async visualizzaAlert() {  
  const msgBox = await this.alertController.create({  
    header: 'Attenzione',  
    subHeader: 'Nessun check selezionato',  
    message: 'I dati non verranno aggiornati',  
    buttons: ['OK']  
  });  
  await msgBox.present();  
}
```

Il metodo **create()** è un metodo asincrono che esegue la creazione della finestra di dialogo

Il metodo **present()** è anch'esso un metodo asincrono che esegue la visualizzazione della finestra di dialogo.

Per cui il metodo **present()** dovrebbe essere richiamata come callback della **create()** (passandola come secondo parametro al metodo **create()**).

Allo stesso modo l'eventuale codice successivo a **visualizzaAlert()** dovrebbe essere eseguito nella callback di **present()**

#### **async / await**

**async / await** sono istruzioni javascript che consentono di scrivere **codice asincrono** come se fosse sincrono.

**async** dichiara che la funzione **visualizzaAlert()** è una funzione asincrona, cioè contenente metodo asincroni.

**await** può essere utilizzato SOLO all'interno di una funzione dichiara **async** e davanti ad un metodo asincrono, trasformandolo in sincrono, cioè bloccante. L'istruzione successiva verrà eseguita soltanto al termine dell'istruzione corrente.

## Finestra di dialogo con pulsanti multipli: **confirm alert**

Rispetto alla precedente **basic alert** consente di restituire una risposta mediante la scelta di un pulsante.

```
async visualizzaAlertConRisposta() {
  const msgBox = await this.alertController.create({
    header: 'Attenzione',
    subHeader: 'Nessun check selezionato',
    message: 'Vuoi abilitare entrambi i generi ?',
    buttons: [
      {
        text: "no",
        cssClass: "secondary",
        role: "cancel",
        handler: () => { console.log("i'am the handler") }
      },
      {
        text: "si",
        cssClass: "secondary",
        handler: () => {
          this.male=true;
          this.female=true;
        }
      },
    ],
  });
  await msgBox.present();
}
```

Il **campo role** impostato al valore CANCEL fa sì che, in corrispondenza di un click sul desktop, la finestra si comporti come se si fosse premuto il pulsante associato alla `role: cancel`, (cioè il pulsante NO) con generazione del relativo handler. Senza la role, la finestra verrebbe chiusa ugualmente ma senza richiamare l'handler().

La chiave `role: "cancel"` può ovviamente essere applicata ad un solo button.

## Creazione di un progetto Tabs

```
ionic start ese02 tabs
```

La pagina **tabs.page.html** contiene un componente **<ion-tabs>** con all'interno una toolbar fissa **<ion-tab-bar>** ancorata sul lato inferiore del componente (`slot="bottom"`) e preposta per essere utilizzata all'interno del componente **<ion-tabs>**

All'interno della toolbar ci sono tre pulsanti di tipo **<ion-tab-button>** anch'essi preposti per essere utilizzati all'interno del componente **<ion-tabs>**

Ognuno di questi pulsanti dispone di un attributo **univoco tab** (di tipo stringa) **che viene utilizzato dal router per navigare attraverso le varie tabs.**

Il **<ion-tabs>** è un componente particolare che dispone di un **<ion-router-outlet>** integrato all'interno del quale verranno visualizzati i componenti selezionati all'interno della **<ion-tab-bar>**.

```

<ion-tabs>
  <!-- <ion-router-outlet> </ion-router-outlet> -->
  <ion-tab-bar slot="bottom">
    <ion-tab-button tab="tab1">
      <ion-icon name="triangle"></ion-icon>
      <ion-label>Tab 1</ion-label>
    </ion-tab-button>
    <ion-tab-button tab="tab2">
      <ion-icon name="ellipse"></ion-icon>
      <ion-label>Tab 2</ion-label>
    </ion-tab-button>
    <ion-tab-button tab="tab3">
      <ion-icon name="square"></ion-icon>
      <ion-label>Tab 3</ion-label>
    </ion-tab-button>
  </ion-tab-bar>
</ion-tabs>

```

In corrispondenza del click su uno dei pulsanti ionic provvederà automaticamente a caricare all'interno dello `<ion-router-outlet>` fittizio il componente indicato all'interno dell'attributo `tab`.

L'attributo `disabled` consente di disabilitare il pulsante `<ion-tab-button>`

### tabs.routing-module.html

All'interno del file di routing del componente `tabs` sono definite le routes children del componente medesimo, cioè:

```

const routes: Routes = [
  {
    path: 'tabs',
    component: TabsPage,
    children: [
      {
        path: 'tab1',
        loadChildren:
          () => import('../tab1/tab1.module').then(m => m.Tab1PageModule)
      },
      {
        path: 'tab2',
        loadChildren: () => import('../tab2/tab2.module').then(m => m.Tab2PageModule)
      },
      {
        path: '',
        redirectTo: '/tabs/tab1',
        pathMatch: 'full'
      }
    ]
  },
  {
    path: '',
    redirectTo: '/tabs/tab1',
    pathMatch: 'full'
  }
];

```

- la **prima route** indica che in corrispondenza della richiesta `/tabs` deve essere visualizzato il componente corrente **TabsPage**.
- Le **route children** indicano il modulo da caricare in corrispondenza della richiesta dei vari path.
- L' **ultima route** indica che in corrispondenza della richiesta `/` deve essere eseguito un redirect alla risorsa `/tabs/tab1`.

### I tre sotto-componenti tab

I tre componenti **tab** sono costituiti da una **intestazione** e da un **contenuto** esattamente come per la pagina blank

```
<ion-header [translucent]="true">
  <ion-toolbar>
    <ion-title>
      Tab 1
    </ion-title>
  </ion-toolbar>
</ion-header>

<ion-content [fullscreen]="true">
  <ion-header collapse="condense">
    <ion-toolbar>
      <ion-title size="large">Tab 1</ion-title>
    </ion-toolbar>
  </ion-header>

  <app-explore-container name="Tab 1 page"></app-explore-container>
</ion-content>
```

Il contenuto di questa pagina è più o meno lo stesso della pagina **blank** con l'aggiunta, all'interno dello `<ion-content>`, di un sotto-componente **app-explore-container**

### Il sotto-componente app-explore-container

Si tratta di un componente custom definito all'interno del progetto e costituito dal seguente codice:

```
<div id="container">
  <strong>{{ name }}</strong>
  <p>Explore <a target="_blank" rel="noopener noreferrer"
    href="https://ionicframework.com/docs/components">UI Components</a>
  </p>
</div>
```

Cioè visualizza un titolo `<strong>` seguito da un collegamento ipertestuale al link indicato.

Il titolo visualizza il contenuto di una variabile **name** definita all'interno del file `.ts` mediante l'utilizzo di un decoratore **Input()** ed iniettata dal chiamante.

```
@Input() name:string;
```

Cioè il chiamante, al momento dell'utilizzo del componente `<app-explore-container>` dovrà 'passargli' una variabile **name** contenente il titolo da visualizzare.

Nel chiamante **name** è scritto senza quadre in quanto a destra c'è un valore diretto.

Il componente presenta infine un file **CSS** sulla base del quale il contenuto viene visualizzato a centro pagina tramite `position:absolute`.

## Creazione di una apk ionic - cordova

Per creare l'apk di un progetto ionic si può utilizzare il comando **ionic cordova** che consente di automatizzare e velocizzare alcune operazioni

Dalla cartella dell'applicazione lanciare il comando **setenv** per l'impostazione delle variabili d'ambiente.

Nel caso di ionic occorre aggiungere la seguente variabile d'ambiente:

```
set _JAVA_OPTIONS=-Xmx512M
```

che serviva per impostare correttamente le dimensioni della memoria.

Con java 1.8 non serve più anzi sembrerebbe creare problemi.

Prima del build occorre installare `npm install -g cordova-res`

Lanciare quindi i comandi cordova:

```
ionic cordova platform add android
ionic cordova build android --prod
```

L'opzione **--prod** serve a settare a true il flag angular di "production configuration" che rende l'apk installabile sul device. Senza questa opzione l'apk non potrà essere installata sul device. Anche con l'opzione **--prod** si tratta però sempre di una apk di debug non adatta per il rilascio su uno store.

Le opzioni **--prod** **--release** servono invece per creare una apk non di tipo "debug" ma di tipo "release" rilasciabile sugli stores. Attenzione però che una apk di tipo "release" deve essere controfirmata con le apposite chiavi, altrimenti non solo non sarà pubblicabile sugli stores, ma non sarà nemmeno installabile sui propri device.

L'apk creata con il build viene salvata all'interno del percorso  
`\platforms\android\app\build\outputs\apk\debug`

## Esecuzione diretta della app

Come nel caso di cordova, anche per le app ionic risulta possibile eseguirla direttamente senza doverla installare su un device. A tal fine occorre :

- Installare `npm install -g native-run`
- Aprire un emulatore oppure collegare un device
- Lanciare uno dei seguenti comandi:

```
ionic cordova run android // build + run su emulatore o device
ionic cordova run android --no-build // solo run (installa soltanto)
```

## Utilizzo dei Plugin

Per l'utilizzo dei plugin occorre preventivamente installare (prima del build)

```
npm install @ionic-native/core
```

dopo di che, per ogni plugin, occorre installare due componenti, il **cordova native code** e il **ionic typescript native code**. Ad esempio, nel caso della fotocamera:

```
// Install Cordova plugin
ionic cordova plugin add cordova-plugin-camera

// Install Ionic Native TypeScript wrapper
npm install @ionic-native/camera
```

---

## Accesso da codice

---

Angular apps can use either Cordova or Capacitor to build native mobile apps. Import the plugin in a `@NgModule` and add it to the list of Providers. For Angular, the import path should end with `/ngx`. Angular's change detection is automatically handled.

```
// app.module.ts
import { Camera } from '@ionic-native/camera/ngx';

...

@NgModule({
  ...
  providers: [
    ...
    Camera
    ...
  ]
  ...
})
export class AppModule { }
```

After the plugin has been declared, it can be imported and injected like any other service:

```
// camera.service.ts
import { Injectable } from '@angular/core';
import { Camera, CameraOptions } from '@ionic-native/camera/ngx';

@Injectable({
  providedIn: 'root'
})
export class PhotoService {
  constructor(private camera: Camera) { }

  takePicture() {
    const options: CameraOptions = {
      quality: 100,
      destinationType: this.camera.DestinationType.DATA_URL,
      encodingType: this.camera.EncodingType.JPEG,
      mediaType: this.camera.MediaType.PICTURE
    }

    this.camera.getPicture(options).then((imageData) => {
      // Do something with the new photo

    }, (err) => {
      // Handle error
      console.log("Camera issue: " + err);
    });
  }
}
```

---

## Pubblicazione di un progetto Ionic su Firebase

---

Dovrebbe essere tutto come per Angular, sostituendo il comando `ng build --prod` con `ionic build --prod`