Guida di Angular 5

Data 1° pubblicazione 18/10/2017
Rilascio 9 del 21 Novembre 2017
Fabio Guerrazzi
https://unplugged855.wordpress.com/
unpluggedmail@gmail.com

Sommario

Capitolo 1

Cosa è Angular

Angular 5 è un framework JavaScript che potenzia in modo considerevole quello che dagli anni 2000 è stato introdotto come Web 2.0 ovvero l'asincronismo AJAX nelle web application. Questa guida ti permetterà di imparare Angular da zero partendo direttamente da Angular 5 senza passare dalle versioni precedenti, della quale verrà fatto riferimento solo per il porting.

Perché Angular?

Ci sarebbe molto da dire sulle scelte che sono state fatte in Google, a partire dalle inutili complicazioni a voler ribattezzare tutto con nuova nomenclatura senza usare neanche uno dei migliaia di termini già ampiamente consolidati dalle precedenti comunità scientifiche. Ma al di là della moda del momento, del fascino dei nuovi strumenti, alla fine Angular ha oggettivamente ragione di essere e assolve brillantemente al suo compito, nessuno meglio di lui. Oggi che il web tende a concentrarsi sul principo SPA (Single-Page-Application) viene facile immaginare web application che in cascata devono aggiornare decine, centinaia, migliaia di porzioni della pagina corrente. Casistiche in cui mi sono imbattuto spesso in ASP.Net e PHP, e si faceva con le classiche chiamate $.ajax con richieste al Back-End e generazione asincrona della porzione di pagina.

> *Chiamata ajax, preparazione, richiesta, pool sul server che inizia a lavorare, parsing, conversione, accesso al db, caricamento dati, conversione in html, rendering pagina, trasferimento dal server al browser, rendering client.*

Fallo un centinaio di volte per una singolo click fatto dall'utente per migliaia di utenti e possiamo andare a casa. Angular è la risposta a questo lavoro: l'asincronismo, il Javascript evoluto che è possibile utilizzare, utilizzo di funzioni reattive, i boundle compatti, tutto questo insieme proiettano le Web Application nella nuova era di gestione efficiente delle risorse di networking probabilmente per alcuni decenni a venire.

Riferimenti e Copyright

Questa guida è liberamente tratta dal libro *Pro-Angular 2* di Adam Freeman, adattato da me alla versione 5, piu il libro *Building m web applications using Angular* di Shravan Kumar Kasagoni, quest'ultimo direttamente scritto per la vers 5. Piu altri articoli interessanti presi in rete, alcune parti derivano direttamente dalla guida ufficiale di Angular, insomma è un mix di quello che ho trovato interessante per imparare ad usarlo. Per me la via piu efficiente per imparare un linguaggio è scrivere cosa ci ho capito. Questo è lo scopo di questa guida, sono le note e gli appunti presi durante il mio corso di tutorial. Sono sicuro che può essere utile ad altri. I libri ovviamente contengono tutte le informazioni dettagliate e sono sicuramente piu utili di questo documento. Il vantaggio di questa guida è essere totalmente gratuita e in italiano. E' comunque severamente vietata la copia o la riproduzione in ogni forma a scopo di lucro, sia del testo che dei sorgenti.

Requisiti minimi per comprendere questa guida

Non sarà sottinteso nulla riguardante Angular, mentre lo sarà per tutto il resto. Ciò significa che è dato per scontato che tu conosca bene come funziona il protocollo http, Javascript, AJAX. Che tu conosca NodeJS non è necessario, basta sapere cosa è e i principi di funzionamento, lo impareremo bene studiando Angular

Note sulle versioni

Attualmente (novembre 2017) è ufficiale la versione 5 uscita l'8 novembre 2017.
Si parla di un piano di uscita della 6 a marzo 2018, ma al momento è tutto in fase embrionale.
Quello che è importante sapere, al momento, è sul naming convention e un breve accenno alla sua storia.
Angular nasce come AngularJS 1 e nel giro di un paio di anni si evolve nella versione 2. Dalla 1 alla 2 non si puo propriamente parlare di versione ma di una completa riscrittura del framework. Questo ha portato non pochi problemi a chi aveva iniziato a distribuire applicazioni in produzione con la vers 1. La 2 era tutto un altro linguaggio. Dalla 2 alla 4 e poi la 5 (non ho notizie di versione 3) fortunatamente non si è trattato di riscrittura ma di vero e proprio aggiornamento, il core è rimasto lo stesso.
Dalla 5 alla 6 idem, saranno solo nuove implementazioni e pochi miglioramenti strutturali.
Gli standar di denominazione di Angular si sono evoluti in questo modo
E' nato AngularJS 1 e poi AngularJS 2. Dalla versione 4 è stato rimosso il JS e il nome è Angular, indipendentemente dal numero di rilascio. Anche questo creava un problema, ormai risolto, di upgrade in quanto le versioni angular erano hard-coded dentro le librerie e aggiornarlo richiedeva molti interventi manuali su codice di sistema.
Da Angular 4 tutti i riferimenti sono semplicemente riferiti ad "Angular" e non Angular-<numero> come avveniva in precedenza

Da Angular 2 ad Angular 5

Per chi aveva gia familiarita con Angular 2 (io no), nella 5 sono spariti i js compilati dal ts, che in effetti erano un po superflui. Se non avevi mai visto A2 sappi che per ogni ts il compilatore generava .js. Non ci sono piu, ecco. O meglio, ci sono ma sono generati a runtime e cancellati al termine dell'esecuzione, non si vedono nel progetto come prima.

Dal rilascio di Angular 2, il pacchetto **angular-cli** è diventato il modo standard per creare e gestire progetti angolari. **angular-cli** è il tool per semplificare il setup e il controllo dei packages. Il rilascio della versione 1 di angular-cli è ora parte integrante di Angular.

Il vantaggio di utilizzare angular-cli è quello di semplificare il processo di installazione, il che significa che non è necessario gestire manualmente l' insieme di pacchetti NPM, la configurazione del compilatore di TypeScript o impostare un server HTTP di sviluppo. Gli step per la preparazione di una nuova applicazione Angular 4 per la distribuzione è decisamente molto più semplice.
Lo svantaggio è che non vedrete i dettagli di basso livello di come funziona il binding tra i moduli base e il modo in cui sono uniti i diversi pezzi (a meno che tu non vada a controllare, ovviamente).

Da dove si parte

leggere una app angular può essere il primo step per capire chi è. Spulciando le varie guide in inglese ho trovato molto molto pertinente una citazione:

> *"se hai bisogno di una webapp che ti dia risultati rapidi usa jQuery. Se hai tempo per analizzare e controllare il rendering html vale la pena di imparare Angular"*

Questa dice molte cose. Angular non è di facilissimo impatto, non consente aplicazioni on the fly, piuttosto richiede una attenta strategia di programmazione e strutturazione delle app, anche se sono semplicissime da un punto di vista utente. A favore c'è che ti permette di fare **SPA** (*Single Page Application*) nel modo piu efficiente possibile

Partire da una app esistente, cosa fa?

Tutto ebbe inizio da package.json :)
Se il progetto è stato creato con li CLI ng new xxx dovresti avere un `main.ts`.
Se è stata fatta in Angular lo vedi perche ci sono questi due files: `.angular-cli.json e src\main.ts`
Se non ci sono questi files apri i json alla ricerca di un app name, root path. Ti solito puoi trovare \src\app.js, app.ts, main.ts

Una volta trovato l'entry point dell'app, cerca di capire cosa esegue e cosa renderizza come home page.
In questi files dovresti trovare un riferimento a app.module.ts, che è il core dell'applicazione. In questo file sono dichiarati tutti i componenti e le librerie utilizzate dal programma.

Apri quindi `app.module.ts` e dovresti trovare questa dichiarazione `bootstrap:[AppComponent]`
Apri l'index.html se cè e apri quindi i files componenti `app.component.ts` e `*.html`
Se c'è un routing vedi il bootstrap su quale componente lo fa, oppure sono strutturati scaffolding tipici MVC cerca i controllers e moduls.

Capire bene Observables e lib RxJS.

Un mucchio di lavoro da fare per imparare angular riguarda capire bene cosa è **Observable** e come si usa. E' vitale capire come funziona e fai tutti i tentativi possibile finche non ti senti sicuro con **RxJS**, perche rappresentano il vero core di Angular. L'argomento sarà trattato a diversi livelli e con più esempi possibile.

I Wizard e tools di Angular

Una volta presa confidenza con l'ambiente considera queste librerie PrimeNG or ValorSoft come standard tra i tuoi strumenti di sviluppo. Provale e vedrai che li integri nel tuo ambiente.

Zone

NgZone è abbastanza easy to use . capire cosa è aiuta

Code structuring

1. **Condividi i Moduli** — Sforzati di usare moduli condivisi. Crea un modulo che importi ed esporti tutti i moduli comuni e importali negli altri moduli. Eviti di reimportarli ovunque
2. **Global vs local CSS** — metti le istruzioni CSS a livello di app anziché a livello di applicazione se dovesse essere ripetuto
3. **Tema SCSS** — Quando utilizzi un preprocessore CSS, definisci sempre un file che usi variabili relative al colore, alla dimensione del carattere, ecc. Ti aiuterà quando devi cambiare il tema.
4. **Ereditarietà Typescript** — Cerca di utilizzare l'ereditarietà Typescript. Se disponi di alcune funzionalità che potrebbero essere richieste in diversi componenti, crea un componente base con

funzionalità comuni in modo che tutti gli altri componenti eventualmente possono semplicemente estenderlo.

5. **Usa i Servizi** — Sforzati di tenere separati view e service. Nel componente dell'interfaccia utente tieni solo il codice deputato alla visualizzazione e delega servizi per effettuare chiamate al back-end.

La Prima Applicazione Angular5

La via più breve per imparare Angular è procedere per esempi. In questo capitolo esploreremo come impostare l'ambiente di sviluppo di base con Angular, con un minimo di static mock-up (dati di esempio) e applicheremo le basi per creare una tipica dynamic web application. Piu avanti vedremo come creare una applicazione piu sofisticata e completa, ma per il momento è sufficiente evidenziare quali sono i componenti piu importanti. Anche se non tutto il codice risulterà chiaro immediatamente, è importante prendere familiarità con la sintassi Angular. Alla fine del capitolo tutta la struttura risulterà chiara e lineare, gli step di questa prima fase servono comprendere il flusso di dichiarazione dei componenti

La nostra app sarà composta da una pagina con un campo di input e un pulsante Add, piu' l'elenco dei dati inseriti. L'utente rimuoverà gli elementi cliccando su un ceckbox della lista o aggiungera un nuovo item premendo Add. Il tutto avverrà in modo dinamico senza ricaricare nessuna pagina

Preparazione dell'ambiente di Sviluppo

Primo step è l'installazione di **NodeJS** e NPM da https://nodejs.org/it/download .

Scarica e installa **Visual Studio Code** da https://code.visualstudio.com/download , che consiste nell'IDE cross platform di Microsoft.

Per chi non conoscesse lo strumento, è il Visual Studio di NodeJS (non solo ma ok, usa questo). Comprende da editor, intellisense, debug, search, package management

Infine il browser, è sufficiente che sia presente Chrome sul computer. In questa guida ci riferiremo a sviluppo di un progetto su ambiente Windows, e verranno omessi tutti i comandi corrispondenti per Ubuntu

Sia chiaro che NodeJS è un componente server ma basandosi su script client javascript per funzionare ha bisogno del CLR di un browser, ovvero un interprete javascript.

Essendo un linguaggio script non vi stupite se nel corso dello sviluppo delle vostre app troverete difficoltà ad interpretare cosa è server e cosa è client, perche non è cosi scontato. Il linguaggio sottostà a tutte le regole client, ad esempio non si puo salvare nulla sul filesystem, benchè tutto l'asincronismo delle richieste e response sono chiaramente caratteristiche server.

Creazione e preparazione del Progetto

Posizionati sulla cartella radice dei tuoi progetti (es c:\src_angular e digita il comando ng new seguito dal nome della nuova applicazione, in questo caso la nostra prima app si chiamerà **todo**

ng new todo

Il sistema scaricherà tutti i pacchetti piu recenti e li posizionerà nella nuova cartella di progetto todo

PS C:\src_angular\ > ng new todo

```
 create todo/e2e/app.e2e-spec.ts (286 bytes)
 create todo/e2e/app.po.ts (208 bytes)
 create todo/e2e/tsconfig.e2e.json (235 bytes)
 create todo/karma.conf.js (923 bytes)
 create todo/package.json (1309 bytes)
 create todo/protractor.conf.js (722 bytes)
 create todo/README.md (1020 bytes)
 create todo/tsconfig.json (363 bytes)
 create todo/tslint.json (2985 bytes)
 create todo/.angular-cli.json (1281 bytes)
 create todo/.editorconfig (245 bytes)
 create todo/.gitignore (516 bytes)
 create todo/src/assets/.gitkeep (0 bytes)
 create todo/src/environments/environment.prod.ts (51 bytes)
 create todo/src/environments/environment.ts (387 bytes)
 create todo/src/favicon.ico (5430 bytes)
 create todo/src/index.html (291 bytes)
 create todo/src/main.ts (370 bytes)
 create todo/src/polyfills.ts (2667 bytes)
 create todo/src/styles.css (80 bytes)
 create todo/src/test.ts (1085 bytes)
 create todo/src/tsconfig.app.json (211 bytes)
 create todo/src/tsconfig.spec.json (304 bytes)
 create todo/src/typings.d.ts (104 bytes)
 create todo/src/app/app.module.ts (314 bytes)
 create todo/src/app/app.component.html (1120 bytes)
 create todo/src/app/app.component.spec.ts (986 bytes)
 create todo/src/app/app.component.ts (207 bytes)
 create todo/src/app/app.component.css (0 bytes)
Installing packages for tooling via npm.
Installed packages for tooling via npm.
Successfully initialized git.
Project 'todo' successfully created.
```

Perchè NodeJS e Angular funzionino devono avere come configurazione di base il file package.json in cui sono presenti tutte le dichiarazioni dei pacchetti necessari al runtime.

Ispezioniamo il file package.json per vedere quali versioni ci ha installato.

Al package generato dall angular-CLI ci va aggiunto il bootstrap e altri pacchetti che non sono inclusi come predefinito. Aggiungi queste linee sotto zone.js nelle declarations

```
    "bootstrap": "4.0.0-alpha.4",
    "font-awesome": "4.7.0",
    "web-animations-js": "^2.3.1",
    "systemjs": "^0.19.40",
```

Il package.json finale che ci serve assomigliera a questo

```
{
```

```
  "name": "todo",
  "version": "0.0.0",
  "license": "MIT",
  "scripts": {
    "ng": "ng",
    "start": "ng serve",
    "build": "ng build",
    "test": "ng test",
    "lint": "ng lint",
    "e2e": "ng e2e"
  },
  "private": true,
  "dependencies": {

    "@angular/animations": "^5.0.0",
    "@angular/common": "^5.0.0",
    "@angular/compiler": "^5.0.0",
    "@angular/core": "^5.0.0",
    "@angular/forms": "^5.0.0",
    "@angular/http": "^5.0.0",
    "@angular/platform-browser": "^5.0.0",
    "@angular/platform-browser-dynamic": "^5.0.0",
    "@angular/router": "^5.0.0",
    "core-js": "^2.4.1",
    "rxjs": "^5.5.2",
    "zone.js": "^0.8.14",
     "bootstrap": "4.0.0-alpha.4",
     "font-awesome": "4.7.0",
     "web-animations-js": "^2.3.1",
     "systemjs": "^0.19.40"
  },
  "devDependencies": {
    "@angular/cli": "1.5.0",
    "@angular/compiler-cli": "^4.2.4",
    "@angular/language-service": "^4.2.4",
    "@types/jasmine": "~2.5.53",
    "@types/jasminewd2": "~2.0.2",
    "@types/node": "~6.0.60",
    "codelyzer": "~3.2.0",
    "jasmine-core": "~2.6.2",
    "jasmine-spec-reporter": "~4.1.0",
    "karma": "~1.7.0",
    "karma-chrome-launcher": "~2.1.1",
    "karma-cli": "~1.0.1",
    "karma-coverage-istanbul-reporter": "^1.2.1",
    "karma-jasmine": "~1.1.0",
    "karma-jasmine-html-reporter": "^0.2.2",
    "protractor": "~5.1.2",
    "ts-node": "~3.2.0",
    "tslint": "~5.7.0",
    "typescript": "~2.3.3"
  }
}
```

Bootstrap per Angular 5

Per utilizzare correttamente i componenti js di bootsrap occorre installare la parte ‘plugin’

https://ng-bootstrap.github.io/#/getting-started

Setup

```
npm install --save @ng-bootstrap/ng-bootstrap
```

app.module.ts

```
import {NgbModule} from '@ng-bootstrap/ng-bootstrap';
@NgModule({
..
  imports:
  [NgbModule.forRoot()],
..
```

Check dei pacchetti NPM

In AngularJS 2 l'installazione dei pacchetti andava fatta a mano. Con ng new non serve, ma se qualcosa dovesse andare storto per conflitti o versioni sbagliate, per aggiornare la nostra cartella di lavoro con tutto l'ambiente è sufficiente posizionarsi sulla cartella attraverso un prompt di DOS o PowerShell e digitare il comando

NPM install

Tutti i pacchetti verranno scaricati e copiati nella cartella di sistema /node_modules

Check configurazione TypeScript

Come NPM install in AngularJS 2, la configurazione del TypeScript andava aggiunto a mano mentre la 4 con ng new la crea da solo. Questo è il mio file tsconfig.json

```
{
  "compileOnSave": false,
  "compilerOptions": {
    "outDir": "./dist/out-tsc",
    "sourceMap": true,
    "declaration": false,
    "moduleResolution": "node",
    "emitDecoratorMetadata": true,
    "experimentalDecorators": true,
    "target": "es5",
    "typeRoots": [
      "node_modules/@types"
    ],
    "lib": [
      "es2017",
      "dom"
    ]
  }
}
```

Bene. Se tutto è andato a buon fine abbiamo NodeJS e Angular 4 installati pronti per poter gestire il codice della nostra applicazione web

Questi sono i files di base creati da **Angular-cli** ng new

La root folder (nel nostro caso todo)

package.json	L'elenco di tutti i pacchetti del motore Angular, NodeJS e vari plugin
package.lock.json	File package dettagliato generato dal compilatore
tsconfig.json	La configurazione di TypeScript
favicon.ico	L'icona della web app
.angular-cli.json	Tutte le definizioni per l'app: nome, file di avvio main.ts, pagina html, cartelle.
.editorconfig	Questo file aiuta gli sviluppatori a definire e mantenere stili di codifica coerenti tra diversi editor e IDE. esso consta di un formato di file per la definizione di stili di codifica e una raccolta di plugin di editor di testo che consentono agli editor di leggere il formato di file e di aderire agli stili definiti. I file EditorConfig sono facilmente leggibili e funzionano bene con i sistemi di version controls.
.gitignore	Ignore file di Git, i files specificati in questo files non verranno inviati al repository

karma.conf.js	File di configurazione di Karma. Karma è l'ambiente low level di test, compatibile con continuous delivery e protractor
protractor.conf.js	File di configurazione di protractor, protractor è l'ambiente di esecuzione e test end-to-end di Angular 4. Usa questo file per modificare l'end point o la porta localhost:4200
Readme.md	File Readme. Contiene informazioni sull'esecuzione, test, delivery
tslint.json	Configurazione tslint. TSLint è uno strumento di analisi statica estensibile che controlla il codice TypeScript per errori di leggibilità, manutenzione e funzionalità. È ampiamente supportato dai moderni sistemi di editing e build e può essere personalizzato con le tue regole, configurazioni e formattazioni.

La cartella \src

node_modules	La cartella di sistema NodeJS completa dei pacchetti dichiarati in package.json (176MB circa)

main.ts	Entry point. Qui vencono inclusi i moduli, pacchetti, librerie, cartelle
	Defaults:
	`import { AppModule } from './app/app.module';`
	`import { environment } from './environments/environment';`

polyfill.ts	ts di sistema per l' inclusione dei plugin esterni
styles.css	File di stile inizialmente vuoto che fa l'override al bootstrap
test.ts	Entry point unit test
tsconfig.app.json	Configura TypeConfig per l'app corrente
tsconfig.spec.json	Ulteriore Configurazione di TypeConfig per esigenze specifiche
typing.d.ts	Data Type NodeModule (default string), definizione modulo SystemJS

La cartella \environment

environment.ts	Opzioni di esecuzione test o produzione. Qui puoi aggiungere qualsiasi altra variabile di configurazione (username, key, secretkey, dbhost, ecc)
environment.prod.ts	Copia di prod di environment.ts. viene elavorato solo l'altro, questa è la copia

La cartella \assets (cartella di risorse, immagini, video, mp3)

.gitkeep	File vuoto (non so cosa sia ☺)

La cartella \app

app.module.ts	La nostra applicazione (richiamata da main.ts) contenente tutte le definizioni dei nostri componenti
app.component.html	contiene le direttive per la sostituzione dei valori nel tag presente in index.html
app.component.ts	contiene la logica per il binding dei dati da portare sull'html
app.component.spec.ts	contiene la logica per il binding dei dati da portare sull'html
app.component.css	Foglio di di stile (inizialmente vuoto) del componente

Creazione **`index.html`** e avvio WebApp

incolla il codice sottostante, questa sarà la home page dell'app

```
<!DOCTYPE html>
<html>
<head>
    <title>My app example</title>
    <meta charset="utf-8" />
    <link href="node_modules/bootstrap/dist/css/bootstrap.min.css" rel="stylesheet" />
</head>
<body class="m-a-1">
    <h1>Hello World!</h1>
</body>
</html>
```

A questo punto digita il comando

```
npm start
```

e vedrai avviarsi il server in ascolto sulla porta 4200
avvia il browser e apri l'url della app:

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

A video dovrebbe comparire una pagina bianca con questo contenuto

Hello World!

Questo significa che la tua prima applicazione Angular 5 sta funzionando. E' bastato creare un nuovo progetto e definire la pagina index.html.
Al momento non abbiamo nessuna funzionalità Angular in esecuzione ma ci siamo assicurati che il motore NodeJS e le dipendenze dichiarate stanno rispondendo.

Aggiunta del contenitore del template
Adesso aggiungeremo un tag nostro **`<todo-app>`** al file index.html che fungerà da container del template che definiremo piu avanti. Il tag ce lo possiamo inventare perche ci riferiremo ad esso per informare Angular dove fare il rendering dei dati che vogliamo visualizzare.

In Asp.Net è il tag `<aps:ContentPlaceHolder></aps:ContentPlaceHolder>` della master page. L'aspx che contiene `<asp:Content></asp:Content>` con lo stesso id andrà a sostituire il paceholder

Modifica il body del file index.html

```
<body class="m-a-1">
    <todo-app>Angular placeholder</todo-app>
</body>
```

Vedremo piu avanti come creare il Content Angular. Ora definiamo i dati

Aggiungiamo il Dataset Angular e personalizziamo la nostra lista di prova

Le applicazioni Angular sono tipicamente scritte in TypeScript. TypeScript è il superset di Javascript sviluppato da Microsoft. In parole molto povere è un linguaggio macro che il compilatore interpreta e traduce in semplice js. E' piu compatto di javascript e una volta familiarizzato è piu semplice da leggere.

Per creare un Data Model per l'applicazione ho aggiunto un `file src/app/models/model.ts` (i file di TypeScript hanno estensione *.ts) ed ho aggiunto il codice sottostante

```
export class Model {
    user;
    items;

    constructor() {
        this.user = "Adam";
        this.items = [new TodoItem("Buy Flowers", false),
                      new TodoItem("Get Shoes", false),
                      new TodoItem("Collect Tickets", false),
                      new TodoItem("Call Joe", false)]
    }
}

export class TodoItem {
    action;
    done;

    constructor(action, done) {
        this.action = action;
        this.done = done;
    }
}
```

Nell'elenco ho usato la sintassi literal di oggetto JavaScript assegnando un valore a una variabile globale denominata model. L'oggetto data model dispone di una proprietà user che fornisce il nome utente e un array di items, ognuno con azione e proprietà. Ogni elementi rappresenta un compito nell'elenco delle attività ToDo.

Templates e Components

I templates in angular sono pezzi di html con markup html misto a sintassi Angular che una volta elaborato in accoppiata con il rispettivo component va a sostituire il markup di destinazione.

Per chi ha familiarità in Asp.Net il meccanismo è lo stesso di `<aps:ContentPlaceHolder></aps:ContentPlaceHolder>` della master page. L'aspx che contiene `<asp:Content></asp:Content>` con lo stesso id andrà a sostituire il paceholder.

Il `ContentPlaceHolder` lo abbiamo nella pagina index.html al tag `<todo-app>`
Il `Content` è il template `app.component.html` che vediamo qui di seguito
Il codice `*.cs` è il nostro `app.component.ts` in cui viene specificato il tag di destinazione `<todo-app>` ,il template è il codice ts da eseguire.

Fai molta attenzione a questi due files app.component.* perche costituiscono il Core di Angular. Analizzeremo ogni singola istruzione che mostreremo in questo esempio in modo da esporre quali sono le funzionalità di base e i principi di pattern di Angular. Una volta assimilate le basi l'evoluzione e l'aggiunta di caratteristiche piu avanzate rispetterà sempre questa logica

Utilizzo dei Templates

I Templates di Angular sono i componenti che vengono mappati attraverso Il nome del file che segue le convenzioni standard di denominazione Angular `<folder>.component.html` e che contiene il markup che viene elaborato dal compilatore

Crea il file `app.component.html` nella cartella *src\app*

```
<h3 class="bg-primary p-a-1">{{getName()}}'s To Do List</h3>
<div class="m-t-1 m-b-1">
    <input class="form-control" #todoText />
    <button class="btn btn-primary m-t-1" (click)="addItem(todoText.value)">
        Add
    </button>
</div>
<table class="table table-striped table-bordered">
    <thead>
        <tr><th></th><th>Description</th><th>Done</th></tr>
    </thead>
    <tbody>
        <tr *ngFor="let item of getTodoItems(); let i = index">
            <td>{{i + 1}}</td>
            <td>{{item.action}}</td>
            <td><input type="checkbox" [(ngModel)]="item.done" /></td>
            <td [ngSwitch]="item.done">
                <span *ngSwitchCase="true">Yes</span>
                <span *ngSwitchDefault>No</span>
            </td>
        </tr>
    </tbody>
</table>
```

Il valore dei dati di un modello viene eseguito utilizzando doppie parentesi graffe - `{{ function() }}` e Angular valuta tutto ciò che c'è al suo interno per ottenere il valore da mostrare.
In questo caso l'associazione dei dati dice ad Angular di invocare una funzione chiamata getName che

sostituirà il contenuto dell'elemento h3 sul response. La funzione **getName** la vediamo nel **component.ts** qui sotto

Creazione di un Component

Un componente Angular è soggetto allo stesso standard *naming convention* dei templates ma contiene codice TypeScript, quindi crea **app.component.ts** con il seguente codice

```
import { Component } from "@angular/core";
import { Model, TodoItem } from "./model";

@Component({
    selector: "todo-app",
    templateUrl: "app.component.html"
})
export class AppComponent {
    model = new Model();

    getName() {
        return this.model.user;
    }

    getTodoItems() {
        return this.model.items.filter(item => !item.done);
    }

    addItem(newItem) {
        if (newItem != "") {
            this.model.items.push(new TodoItem(newItem, false));
        }
    }
}
```

In questo file mettiamo il codice per la gestione di un Data Model da elaborare per la visualizzazione sulla pagina.

Le Direttive di Angular

Qua siamo ancora in JavaScript, però il codice si basa su funzionalità nuove e che costituiscono le direttive tipiche di Angular. Sentiremo parlare continuamente di Direttive perché il linguaggio si basa sostanzialmente di parti dichiarative più che puro codice procedurale. Il codice del component TypeScript precedente può essere suddiviso in tre sezioni principali: **Import, Decorator e Class**

Capire l'Import

```
import { Component } from "@angular/core";
import { Model } from "./model";
```

La keyword **Import** è la controparte dell'**export** usata in NodeJS. Se l'export dice al compilatore "*rendi questo elemento visibile all'esterno di questo oggetto*, il (*public class in c#),* l'Import è l'*include* di codice nel contesto corrento (la *Using xxxx di c#)*
La **Import** dice al compilatore di rendere l'elemento del parametro nello scope corrente. Le risorse contenute nell'import diventeranno direttamente visibili nel codice corrente. Lo scope gerarchico della **Import** è prima di tutto la cartella **`node_modules`** e poi la cartella in cui è posizionato il componente in cui è dichiarato.
Il nome del from è una cartella fisica su filesystem ed eventualmente il nome parziale.
Se su filesystem abbiamo
`/model/service.data.login.ts`
`/model/service.data.orders.ts`

Possiamo dichiarare

```
import { DataServices } from "./model/service.data";
```

per riferisi a tutto il dataset
oppure

```
import { DataLogin } from "./model/service.data.login";
```

per importare solo i membri di login

Sintassi:
`Import {"nome richiamabile"} from "<elemento fisico su filesystem>"`

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

Dice al compilatore dove prendere tutte le librerie Angular. @Angular/core se si va a vedere non è altro che una cartella contenuta in node_modules

```
import { Model } from "./model";
```

Dice al compilatore che abbiamo definito un nostro modello. Il punto "." Informa che la root di ricerca è la cartella corrente, /<elemento> è un qualsiasi nome parziale. Attenzione che senza estensione verranno inclusi tutti i files che iniziano con questo nome, quindi per capire model.ts, model.js, modellodiprova.html sono tutti da considerarsi files inclusi

Capire il Decoratore

```
@Component({
    selector: "myapp-app",
    templateUrl: "app/app.component.html"
})
```

La keyword @<nome decoratore> definisce il metadata della classe. In questo caso l'abbiamo chiamata Component sopra nella dichiarazione, si puo usare un nome qualsiasi

`selector` è il tag che abbiamo inserito nella pagina index.html

`templateUrl` è il componente (il template html) che verrà renderizzato al suo interno

Quando viene avviata l'app, il compilatore esegue il pattern dei component associati alla pagina index.htm ed esegue tutti i contenuti sostituendo tutti i tag che incontra

Come si puo vedere Il principio è molto semplice e molto versatile, pensate e immaginate centinaia o migliaia di tag in cascata, è facile intuire la potenza dello strumento.

Piu avanti vedremo altre keyword del decoratore `@Component`

capire la Class

```
export class AppComponent {
    model = new Model();

    getName() {
        return this.model.user;
    }

    getTodoItems() {
        return this.model.items.filter(item => !item.done);
    }

    addItem(newItem) {
        if (newItem != "") {
            this.model.items.push(new TodoItem(newItem, false));
        }
    }
}
```

Queste istruzioni definiscono una classe oggetto denominata `AppComponent` che espone proprietà e metodi. Esse forniscono le funzionalità necessarie per supportare l'associazione dei dati nel modello. Quando viene creata una nuova istanza della classe `AppComponent`, la proprietà model viene impostata su una nuova istanza della classe Model che è accessibile all'esterno.

La funzione `getName()` come si può vedere restituisce una semplice proprietà stringa e va a sostituire il contenuto del tag dichiarato tra doppie graffe nell'html

```
<h3 class="bg-primary p-a-1">{{getName()}}'s To Do List</h3>
```

La funzione `getTodoItems()` invece estende le funzionalità ad altre caratteristiche Angular
Il primo è l' espressione `*ngFor`, il ciclo for in sintassi angula, che viene utilizzato per ripetere una regione di contenuto per ogni elemento di una matrice. l'espressione viene applicata ad un attributo di un elemento, come questo:

```
..
    <tr *ngFor="let item of getTodoItems(); let i = index">
..
```

`*ngFor ="let <nome obj> of <array>; let <contatore> = index"`
Ovvero metti in item l'array risultante da getTodoItems()
`let <oggetto> of` definisce l'oggetto singolo, `array` l'array di oggetti. Il `contatore` è facoltativo

nell'oggetto Item avremo il singolo elemento dell'array, nell'eventuale contatore abbiamo in numero di elemento con base 0

nel nostro esempio avremo quindi, nella prima colonna il numero, e nella seconda il nome dell'azione

```
..
            <td>{{i + 1}}</td>
            <td>{{item.action}}</td>
..
```

L'espressione `[ngSwitch]` è un'istruzione condizionale utilizzata per inserire diversi set di elementi nel documento basato su un valore specificato, che in questo caso è la proprietà `item.done`.

```
..
            <td [ngSwitch]="item.done">
                <span *ngSwitchCase="true">Yes</span>
                <span *ngSwitchDefault>No</span>
            </td>
..
```

Nascosto dentro l'elemento td sono due elementi di span che sono stati annotati con `*ngSwitchCase` e `*ngSwitchDefault` e sono equivalenti al caso e alle parole chiave predefinite di un normale blocco di switch JavaScript.

`*ngSwitch` fa si che l'elemento del primo span viene aggiunto al documento solo quando il valore della proprietà `item.done` è true e la l'elemento del secondo span viene aggiunto al documento quando item.done è falso.

Data binding bidirezionale

Una delle caratteristiche interessanti di Angular è che nell'associazione dei dati alla pagina non si ha solo in una direzione, cioè dai dati alla pagina, ma è anche possibile aggiornare i valori dalla pagina al database. Questa tecnica si chiama two-ways data binding.

```
            <td><input type="checkbox" [(ngModel)]="item.done" /></td>
```

L'espressione `ngModel` nell'esempio crea un legame bidirezionale tra il valore (`item.done` in questo caso) e un elemento del form (checkbox). Quando salvi le modifiche, vedrai una nuova colonna che contiene i checkboxes della tabella. Il valore del checkbox è impostato con item.done, come una semplice relazione monodirezionale read-only, ma quando l'utente spunta il checkbox Angular risponde aggiornando la proprietà del modello corrispondente, assolutamente in tempo reale.

Filtrare i dati

I checkboxes consentono di aggiornare il modello di dati e il passo successivo è quello di rimuovere gli elementi dalla pagina. Questo è quello che vogliamo che accada in questa applicazione di esempio e per fare ciò dobbiamo filtrare l'array in uscita dei soli elementi contrassegnati come fatti (done = true) nella funzione `getTodoItems()`

```
        return this.model.items.filter(item => !item.done);
```

Se torniamo al `model.ts` possiamo vedere come sono strutturati i dati di mock che abbiamo usato per l'esempio, qui hardcodati. In una applicazione reale questa array saranno in restituzione di una call WebAPI o di un accesso a un db fisico. Nell'esemio abbiamo l'oggetto con valore inizate done=false

```
        this.items = [new TodoItem("Buy Flowers", false),
                      new TodoItem("Get Shoes", false),
                      new TodoItem("Collect Tickets", false),
                      new TodoItem("Call Joe", false)]
```

Per filtrare i dati è sufficiente insirire la keyword `filter` all'oggetto array e la esprimere la condizione all'interno delle parentesi

Aggiunta del campo di Input e Add Button

Il passo successivo consiste nel costruire le funzionalità di base per consentire all'utente di creare nuovi elementi da eseguire e di memorizzarli nel modello di dati. Vediamo il codice per aggiungere un input field in angular

```
<div class="m-t-1 m-b-1">
    <input class="form-control" #todoText />
    <button class="btn btn-primary m-t-1" (click)="addItem(todoText.value)">
        Add
    </button>
</div>
```

Come si puo vedere l'id del tag input si può definire semplicemente aggiungendo il carattere # prima del nome `todoText` . Con questo id si farà riferminento all'interno del controllo Button

In questo esempio come evento click del button (semplicemente messo tra parentesi) si vuole invocare il metodo additem presente nel ts.

```
    addItem(newItem) {
        if (newItem != "") {
            this.model.items.push(new TodoItem(newItem, false));
        }
    }
```

al quale viene passato il valore del tag di input nella variabile newItem ed eseguito un push all'array del data model.
Semplice, lineare ☺

Mettere insieme l'intera applicazione

A questo punto abbiamo i tre pezzi richiesti per costruire una semplice applicazione Angular:
un data model `model.ts`, un template `app.component.html` e un componente `app.component.ts` .
Ora vanno messi insieme per creare l'applicazione. Per fare in modo che il compilatore sappia dove prendere i pezzi useremo un modulo di bind Angular TypeScript, che per convenzione è sempre lo stesso in

tutte le app. In pratica in questo file vengono dichiarate le variabili richiamabili per indirizzare gli elementi fisici che abbiamo visto finora (html, decorator, class)

Creiamo **app.module.ts** nella cartella src\app. Questo file è a livello di applicazione, ce ne sarà uno solo per tutto il sito web, quindi come è facile intuire conterrà tutte le definizioni delle pagine e i vari componenti che costituiscono tutta l'app. Qui abbiamo una sola pagina, espressa da app.component* e la rendiamo accessibile dal compilatore qui dentro:

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

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

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

analizziamo le linee di codice

ngModule

Le prime tre import informano il compilatore dove prendere i componenti di sistema Angular e solitamente riportiamo sempre questi , di qualsiasi applicazione si tratti, consideriamola parte standard di base di una app angular

l'import

```
import { AppComponent } from "./app/app.component";
```

informa il compilatore che utilizziamo un nostro component al quale faremo fare qualcosa per soddisfare le nostre esigenze

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

@NgModule definisce la configurazione del decoratore
Imports: quali sono i componenti di sistema separati da ,
Declarations: nome del dichiaratore (in questo caso ne abbiamo uno solo ed è il nostro app.component)
Bootstrap: chi parte per primo all'avvio dell'app (in questo caso ne abbiamo uno solo ed è il nostro app.component)

```
export class AppModule { }
```

infine l'export informa il compilatore con quale attributo o nome viene visto all'esterno del modulo tutto quello che abbiamo incluso qui

Una applicazione Angular ha infine bisogno del suo entry point, il suo bootstrap, il codice che deve essere eseguito all'avvio della app.
A questo scopo creiamo un file **`main.ts`**, che contiene le direttive di inizializzazione

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

Anche se qui vediamo una applicazione eseguita su di un browser web, Angular è progettato per lavorare su una svariata gamma di ambienti. Le istruzioni di codice nel file **`main.ts`** selezionano la piattaforma che verrà utilizzata e carica il modulo di root, che è il punto di ingresso dell'applicazione.
Il TypeScript che vediamo scritto qua è lo standard adottato per una web app. Anche se ovviamente si possono usare nomi variabili è consigliabile utilizzare AppModule + app.module e platformBrowserDynamic per semplicità di lettura da parte di altri sviluppatori

Alla fine è semplice no? Abbiamo implementato una pagina html, un file di codice ts e un template. model.ts fa da data layer e **`app.module.ts`** e **`main.ts`** fungono da bootstrap di avvio dell'app.

Cross Platform

Vale la pena accennare qualcosa sul cross platform di Angular. La potenzialità dell'ambiente di sviluppo Angular è che se un domani qualcuno si inventa una nuova piattaforma e definisce i propri componenti di runtime, come ha fatto **Ionic** ad esempio, basta che rilasci la cartella NPM dove è posizionato e basterà richiamarlo qui con l'import per avere a disposizione tutto l'engine che lavorerà sullo strato hardware di destinazione

Esecuzione dell'applicazione

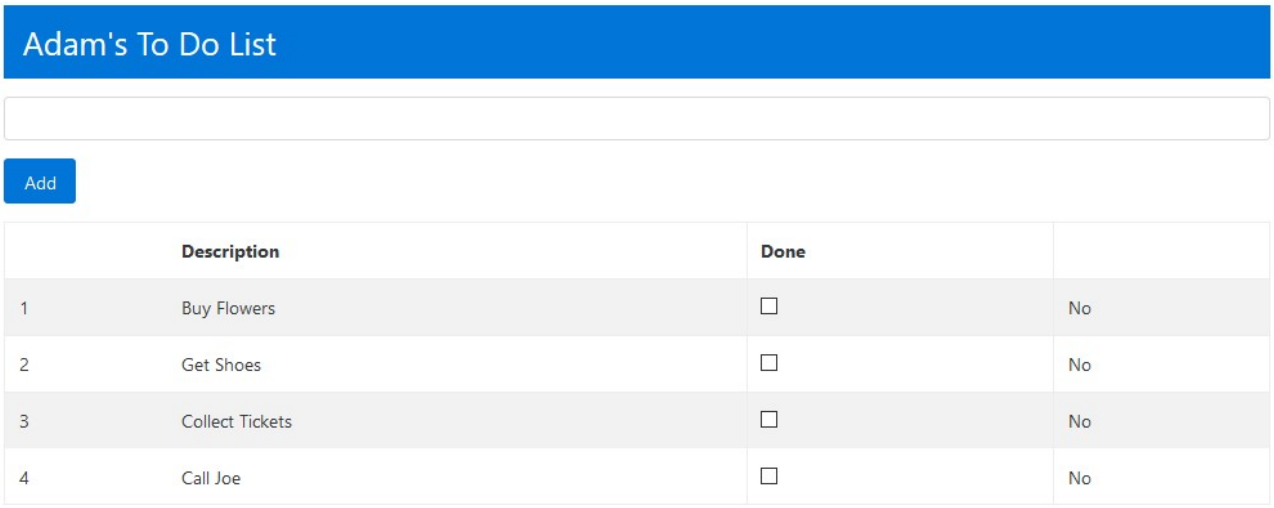
Per eseguire l'app basta posizionarsi sulla root \todo e digitare il comando
`npm start`

dovreste vedere su console

```
PS C:\0\websites\Angular\test\todo> npm start

> todo@0.0.0 start C:\0\websites\Angular\test\todo
> ng serve

** NG Live Development Server is running on http://localhost:4200 **
Hash: 866e1fb3d8ad5cc3dfdd
Time: 6294ms
chunk    {0} polyfills.bundle.js, polyfills.bundle.js.map (polyfills) 226 kB {4} [initial]
[rendered]
chunk    {1} main.bundle.js, main.bundle.js.map (main) 5.24 kB {3} [initial] [rendered]
chunk    {2} styles.bundle.js, styles.bundle.js.map (styles) 9.77 kB {4} [initial] [rendered]
chunk    {3} vendor.bundle.js, vendor.bundle.js.map (vendor) 2.48 MB [initial] [rendered]
chunk    {4} inline.bundle.js, inline.bundle.js.map (inline) 0 bytes [entry] [rendered]
```

```
webpack: Compiled successfully.
```

E sul localhost:4200 vedrete questo:



Eseguire online il sorgente su http://plnkr.co/

Per testare i pezzi di codice al volo online o provare intere app ho usato plnkr.com.
Non funziona, ma se inizializzi una nuova ng new xxxxx e ci innesti questo codice funziona tutto. Non è chiaro il motivo, penso che non lo so usare io ma ho fatto l'upload pari pari dei sorgenti che in locale funzionano e qui no. Lo risolverò, intanto i sorgenti sono li

Utilizzare SystemJS in una app Angular 5

Per fare in modo che TypeScript, NodeJS e Angular interpretino tutti gli standard JavaScript (ES2015, AMD, CommonJS) dobbiamo utilizzare il pacchetto **systemJS**

Installa il pacchetto **systemJS**

```
npm install systemjs
```

Aggiungi il pacchetto a package.json sotto dependancies

```
    "systemjs": "^0.19.40",
```

E le dichiarazioni nel file index.html

```
  <script src="node_modules/systemjs/dist/system.src.js"></script>
  <script> System.config({}); System.import('src'); </script>
```

perche ci serve SystemJS?

Non ci serve, perche abbiamo deciso di utilizzare WebPack nella nostra app, ma averlo installato risulterà utile in futuro e il motivo è il seguente.
Con SystemJS puoi caricare un modulo js senza doverlo dichiarare a livello di applicazione e portarlo dietro ovunque. Se ti serve una funzione al volo (visualizza un PDF) solo in un contesto o condizione particolare, SystemJS puoi caricaricare il modolo. Utilizzare la funzione e muore li, scaricato il componente non esiste piu.

Capitolo 2

Data Binding e Direttive Angular

Per Data Bindings si intendono quelle espressioni che vengono elaborate per produrre html dinamico. E' il sistema che Angular usa per associare i componenti HTML al suo corrispondente TypeScript. Contengono espressioni Javascript anche semplici. Bisogna fare attenzione a non complicare la logica e cercare di tenere piu compatto possibile il codice. Il DataBinding è una delle caratteristiche fondamentali di Angular, il loro uso deve essere costante e continuo.

Binding Target o destinazione delle associazioni

In Angular quando ci riferiamo a elemento Host si untende il tag sull'html, che puo essere un <Div> un <Tr> ecc ecc. L'associazione di Angular e html per creare html dinamico avviene tramite direttive di Binding

Le principali direttive native di Angular sono queste

ngClass	Direttiva per assegnare una classe all'host
ngStyle	Direttiva per assegnare una regola di stile all'host
ngIf	Direttiva per eseguire valutazioni condizionali per il rendering dell'host
ngFor	Direttiva che esegue un ciclo for next per il rendering dell'host
ngSwitch **ngSwitchCase** **ngSwitchDefault** **ngTemplateOutlet**	Direttiva che esegue un case Switch per il rendering di blocchi html nell'host

Quando Angular elabora la destinazione del databinding inizia controllando se corrisponde a direttiva. La maggior parte delle applicazioni si baserà su un mix di direttive native di Angular e altre personalizzate.Le Direttive native iniziano con nome ng e sarebbe opportuno non crearne di proprie che iniziano con lo stesso prefisso.

*ngIf

La direttiva ngIf è utilizzata per aggiungere o rimuovere elementi dal DOM in modo dinamico

```
<element *ngIf="condition"> content </element>
```

Come si vede non è altro che un if condizionale per tag DOM

```
<div *ngIf="isReady">
    <h1>Structural Directives</h1>
    <p>They lets us modify DOM structure</p>
</div>
```

isReady deve essere una variabile boolean presente

*ngFor

L'ngFor esegue un ciclo for su tag DOM su oggetti prevalentemente di tipo JavaScript arrays oppure qualsiasi altro oggetto iterabile.

```
  {{   public frameworks: string[]  = ['Angular', 'React', 'Ember']}}
   <ul>
        <li *ngFor="let framework of frameworks">
            {{framework}}
        </li>
   </ul>
```

*ngSwitch

La direttiva ngSwitch come è facile intuire si comporta come switch case in JavaScript ma viene applicato al DOM.

```
  {{   public selectedCar: string  = 'Ferrari'}}
   <div [*ngSwitch]="selectedCar">
       <template [ngSwitchCase]="'Bugatti'">I am Bugatti</template>
       <template [ngSwitchCase]="'Mustang'">I am Mustang</template>
       <template [ngSwitchCase]="'Ferrari'">I am Ferrari</template>
       <template ngSwitchDefault>I am somebody else</template>
   </div>
```

ngStyle

La direttiva ngStyle è usata quando dobbiamo applicare piu stili inline dinamicamente

```
  {{   public frameworks: string[]  = ['Angular', 'React', 'Ember']}}
   <ul>
        <li *ngFor="let framework of frameworks">
            <p [ngStyle]="getInlineStyles(framework)">{{framework}}</p>
        </li>
   </ul>
```

E nel **component.ts**

```
export class AppComponent {
   getInlineStyles(framework) {
       let styles = { 'color': framework = 'Angular' ? 'red' : 'green','text-decoration': framework= 'Angular' ? 'underline' : 'none'};
       return styles;
   }
}
```

In questo esempio quando il DOM farà il rendering dell'item Angular questo sarà di colore rosso underline e gli altri saranno verdi

ngClass

La direttiva ngClass viene usata quando dobbiamo applicare piu classi in modo dinamico

Nel foglio di stile `component.css`

```
.red {  color: red;  text-decoration: underline; }
.bolder {  font-weight: bold; }
```

Nel template `component.html`

```
    <p [ngClass]="geClasses(framework)">{{framework}}</p>
```

Nel codice `component.ts`

```
    geClasses(framework) {
        let classes = {
            red: framework = 'Angular',
            bolder: framework = 'Angular'
        };
        return classes;
```

```
    }
```

All'elemento Angular verranno applicate le classi **class="red bolder"**

Ad ogni modo, se la classe da applicare fosse condizionale e una sola, la strada è piu semplice, basta una linea nel template

```
    {{public isThisRed = framework = "Angular"? true:false}}
    <p [class.red]="isThisRed">{{framework}}</p>
```

È sufficiente specificare [class.<nome classe>] = true/false e per true fa il render di **class="<nome classe>"**

src

La direttiva src serve per sostituire il nome di un file immagine di un controllo img o link

```
    <img [src]="selectedBook.coverImage" alt="cover image"/>
```

One-Way Data Binding

il DtaBinding unidirezionale è utilizzato per generare contenuti dinamici e costituiscono il blocco di base di Angular. Il termine *One-Way* si riferisce al fatto che il flusso di dati scorre in una direzione, che, nel caso del bindings qui descritto, significa che il flusso va dai dati al template html per il rendering.
Ci sono altri tipi di data binding, che vengono utilizzati generalmente nei forms e consistono nel data binding bidirezionale che descriviamo piu avanti. Questi permettono di aggiornare i dati in tempo reale, e quindi il flusso va in entrambe le direzioni

L'uso di parentesi in Angular

Sembra che uno degli errori piu comuni in Angular sia quello di non usare le parentesi nel modo corretto. Intanto vediamo quali sono

[direttiva]="espressione"	Le parentesi quadre dicono ad angular che il databinding va fatto attraverso una direttiva e la valutazione di una espressione
{{epressione}}	Le doppie graffe dicono ad angular che il contenuto deve essere valutato dal risultato dell'espressione
(evento)="espressione"	Le parentesi tonde dicono ad Angular che deve essere valutato un evento dell'host e deve essere eseguita l'azione definita nell'espressione
[(target)]="espressione"	L'accoppiata quadre e tonde (dette anche a banana-box) dicono ad angular che il binding dei dati è bidirezionale e l'espressione deve aggiornare i dati

del target

Quindi attenzione che se si utilizza **ngClass** senza parentesi quadre la direttiva viene tradotta ma non la sua espressione, che risulterà la stringa letterale

```
<div [ngClass]="'p-a-1 ' + getClasses()" >
  Hello, World.
</div>
```

output

```
<div class="'p-a-1 ' + getClasses()" >
  Hello, World.
</div>
```

Bindings

Binding di proprietà

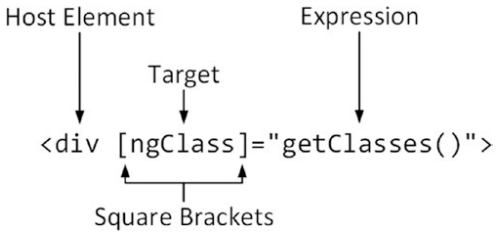
In Angular è possibile dichiarare nell'html delle keyword per indirizzare le proprietà di un elemento.

[proprietà] Suffisso della proprietà

[attr.<nome prop>] Suffisso **attr.** seguito dal nome dell'attributo

[style. <nome prop>] Suffisso style. Seguito dal nome dell'elemento stile CSS

[class. <nome prop>]



Anatomia di un data binding

Binding di attributi

Angular utilizza sempre le proprietà per associare i dati, ma se non esiste una proprietà corrispondente per l'attributo di un elemento, Angular legherà i dati agli attributi. La sintassi di binding attributo inizia con la parola chiave attr seguita dal nome dell'attributo e quindi la assegna alla proprietà della classe Component

Facciamo un esempio con attr. style, class e property seguono le stesse regole

Html semplice. Valore fisso 5 da attribuire a colspan del tag td

```
<td colspan="5"></td>
```

Qui diciamo ad Angular che il colspan deve prendere il valore di una variabile presente nel ts di nome colspanvalue

```
<td [attr.colspan]="colSpanValue"></td>
```

Qui diciamo ad angular che il valore deve essere calcolato nel ts

```
<td [attr.colspan]="getConditionalValue()"></td>
```

Altri esempi

```
<div [ngClass]="'p-a-1 ' + getClasses()" >
  Hello, World.
</div>
```

ts

```
  getClasses(): string {
    return this.model.getProducts().length == 5 ? "bg-success" : "bg-warning";
  }
```

Render:

```
<div class="p-a-1 ng-success" >
  Hello, World.
</div>
```

Interpolazione di Stringhe

Si dice interpolazione di stringhe tutte le espressioni che concatenano testo ad espressioni contenute nel ts. Gli esempi sotto chiariscono bene il contetto

Tag codice: #9878#

L'input con id **txProductName** avrà la proprieta **value** con il nome del prodotto o **"None"** se nullo

```
<div #mydiv [ngClass]="'p-a-1 ' + getClasses()"
     [textContent]="'Titolo del libro: ' + (getBook(9781784396527)?.title || 'None')" >
</div>
<div class="form-group m-t-1">
<label>Inpu Title via Interpolazione stringa:</label>
<input class="form-control" #myTextBox  [placeholder]="getBook(9781784396527)?.title || 'None'" [value]=[sasadsasd asd + pippo] />


</div>



.. nel ts ..
getBook (isbn: number) {
    var selectedBook = this.booksList.filter(book => book.isbn === isbn);
    return selectedBook[0];
  }
```

String e dollaro $

Un altro uso utile delle stringhe è la sostituzione variabili stringhe nella costruzione di una stringa

```
    let giorno:string="lunedi";
    let message = `il ${giorno} resterà chiuso`;
```

Binding di Eventi

Utilizzando la sintassi di binding evento possiamo associare eventi di elementi HTML incorporati, come ad esempio il click, la modifica,il blur ecc, ai metodi di classe Component.

```
<button type="button" (click)="removeMe(this); myForm.control.markAsTouched()">remove</button>
```

Qui diciamo ad angular che al click sul pulsante 'remove' esegue removeMe e un altro comando per la modifica di stato del form (vedremo in dettaglio piu avanti). Come puoi notare qui all'evento non siamo associando jQuery o JavaScript ma sintassi TypeScript. Quello che va tra doppi apici può essere anche un intero script. Ovviamente se si allunga è piu opportuno portare tutto nel component.

Per fare una prova modifica il file **app.component.ts**

```
  public message :  string = 'Angular - Event Binding';
  showMessage(msg) {   alert(msg);  }
```

E nell'app.component.html

```
<h1>{{message}}</h1>
<input type="text" [value]="message" (keypress)="showMessage('premuto un tasto')"/>
```

Oppure possiamo anche legare eventi personalizzati su componenti o direttive

html

```
<h1>{{message}}</h1>
<input type="text" [value]="message" (keypress)="$event"/>
```

ts

```
public message :  string = 'Angular - Event Binding';
showMessage(msg) {
      this.message = msg.target.value
 }
```

Infine, per utilizzare il two-ways-data-binding (spiegato in precedenza)

```
<input type="text" [(ngModel)]="message" />
```

Che fa il read/write della proprietà. Se **`message`** è un dato sul db o proprieta di un modello viene letto e aggiornato in modo bidirezionale, fa tutto angular

Componenti multipli

Di seguito faremo un esempio pratico di un data entry e list form con elenco dati, un pulsante edit, uno add e remove

Nell'esempio abbiamo una tabella contente libri, alcune proprieta e l'immagine di copertina

Mostreremo sulla pagina, prima la lista, e alla selezione il form di dati

Partiamo dai dati

Creiamo una cartella **`\src\app\model`**

E al suo interno il file **`book.ts`**

```
export class Book
{
    isbn: number;
    title: string;
    authors: string;
    published: string;
    description: string;
    coverImage: string;
}
```

e **`mock-book.ts`**

```
import { Book } from './book';

export const BOOKS: Book[] = [  {
        isbn: 9781786462084,
        title: 'Laravel 5.x Cookbook',
        authors: 'Alfred Nutile',
        published: 'September 2016',
        description: 'A recipe-based book to help you efficiently create amazing PHP-based applications with Laravel 5.x',
        coverImage: 'https://d255esdrn735hr.cloudfront.net/sites/default/files/imagecache/ppv4_main_book_cover/B05517_MockupCover_Cookbook_0.jpg'
    },
    {   isbn: 9781784396527,
        title: 'Sitecore Cookbook for Developers',
        authors: 'Yogesh Patel',
        published: 'April 2016',
        description: 'Over 70 incredibly effective and practical recipes to get you up and running with Sitecore development',
        coverImage: 'https://d255esdrn735hr.cloudfront.net/sites/default/files/imagecache/ppv4_main_book_cover/6527cov_.jpg'
    },
    {   isbn: 9781783286935,
        title: 'Sass and Compass Designers Cookbook',
        authors: 'Bass Jobsen',
        published: 'April 2016',
        description: 'Over 120 practical and easy-to-understand recipes that explain how to use Sass and Compass to write efficient, maintainable,
and reusable CSS code for your web development projects',
        coverImage: 'https://d1ldz4te4covpm.cloudfront.net/sites/default/files/imagecache/ppv4_main_book_cover/I6935.jpg'
    }
];
```

Creiamo la nostra home index.html

```
<!doctype html>
<html lang="en">
<head>
```

```
  <meta charset="utf-8">
  <title>MyExamples</title>
  <base href="/">

  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="icon" type="image/x-icon" href="favicon.ico">
  <link href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.min.css"  rel="stylesheet" />
</head>
<body>
    <div style="text-align:center">
        <h1>
          Benvenuto su Angular 4 tutorial book!
        </h1>
        <img width="300"
src="
1IDIzMGw3OC45LTQzLjcgMTQuMi0xMjMuMXoiIC8+CiAgICA8cGF0aCBmaWxsPSIjQzMwMDJGIiBkPSJNMTI1IDMwdjIyLjItLjFWMjMwbDc4LjktNDMuNyAxNC4yLTEyMy4xTDEyNSAzMHoiIC8+CiAgICA8cGF0aCAgZmlsbD0iI0ZGRkZGRiIgZD0iTTEyN
SA1Mi4xTDY2LjggMTgyLjZoMjEuN2wxMS43LTI5LjJoNDkuNGwxMS43IDI5LjJIMTgzTDEyNSA1Mi4xem0xNyA4My4zaC0zNGwxNy00MC45IDE3IDQwLjl6IiAvPgogIDwvc3ZnPg==">
      </div>
      <h3 class="title">Elenco Libri</h3>

      <div class="border-divider">  </div>

      <div class="row">
          <div class="col-md-3">Image</div>
          <div class="col-md-4">Titolo</div>
          <div class="col-md-5">Autore</div>
      </div>

      <books-list>Loading...</books-list>

</body>
</html>
```

Come vedi abbiamo un header con il logo un titolo, una riga di intestazione, e poi il tag <book-list> che conterrà l'elenco dei libri

Ora creiamo il componente per la lista

Creiamo il file globale **`src\app\app.module.ts`**

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

import { BookDetailsComponent } from './book-details/book-details.component';
import { AppComponent } from './app.component';

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

il pezzo di componente html **`src\app\app.component.hml`**

```
<h3 class="title">Books List</h3>
 <div class="border-divider"></div>
 <div class="row">
        <!-- lato sinistro -->
      <div class="col-md-6">

            <ul class="list">

                 <div class="list-item row" *ngFor="let book of booksList" >

                        <div class="row">
                         <div class="col-md-4">
                             <div class="cover-image-container">
                                 <span class="cover-image">
                                     <img [src]="book.coverImage" alt="cover image"/>
                                 </span>
                             </div>
                         </div>
                         <div class="col-md-5">
                             <div class="clear">
                                 <h3 class="book-title">{{book.title}}</h3>
                             </div>
                         </div>
                         <div class="col-md-2">
                                 <h4 class="book-author">{{book.authors}}</h4>
                         </div>
                         <div class="col-md-1">
                          <button class="btn btn-primary" #det (click)="getBookDetails(book.isbn)">View</button>
```

```
                    </div>
                  </div>
              </div>
          </ul>

    </div>

    <!-- lato destro con il dettaglio del libro -->
    <div class="col-md-6">

        <div class="col-xs-12">
          <div  *ngIf="selectedBook" class="row selected-book">
            <div class="col-xs-4">
                <img [src]="selectedBook.coverImage" alt="cover image"/>
            </div>
             <div class="col-xs-8">
                <h3 class="title">{{selectedBook.title}}</h3>
                <p>{{selectedBook.authors}}</p>
                <p>{{selectedBook.published}}</p>
                <p>{{selectedBook.description}}</p>
            </div>
          </div>
       </div>

    </div>
</div>
```

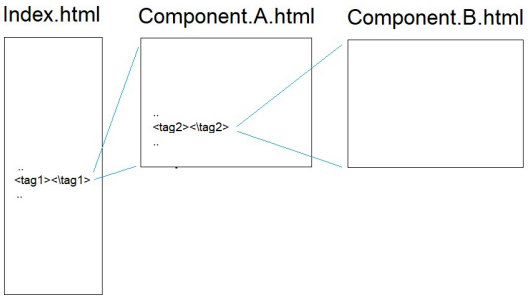
Nell'html abbiamo una cosa interessante, l'*ngIf . Con questa direttiva l'html contenuto nel div viene processato solo se la variabile selectedBook è diverso da null. L'*ngFor è spiegato piu avanti ma è evidente che è un ciclo for classico

Infine il componente td **`src\app\app.component.ts`**

```
import { Component } from '@angular/core';
import { Book } from './model/book';
import { BOOKS } from './model/mock-books'

@Component({
  selector: 'books-list',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})

export class AppComponent {
  booksList: Book[] = BOOKS;
  selectedBook: Book;
  title = 'Elenco Libri';
  getBookDetails (isbn: number) {
        var selectedBook = this.booksList.filter(book => book.isbn === isbn);
         this.selectedBook = selectedBook[0];
     }
}
```

Questa è la web app come si presenta, elenco di 3 libri nel quadrante di sinistra e al click sul pulsante view mostra i dettagli nel quadrante di destra

Componenti multipli



Mostreremo ora come utilizzare piu di un componente in cascata (un tag di Index usa il template comp.A.htm, un tag di comp.A.html usa comp.B.html, ecc).

Nella nostra app dei libri aggiungiamo una cartella book-details e quindi
Posizionati sulla cartella src\app e digita il comando
`ng generate component book-details`

Angular crea la cartella `book-details` e ci crea i files precompilati
`Book-details.component.ts`
`Book-details.component.html`
`Book-details.component.css`

E da solo modifica sulla cartella superiore **`l'app.module.ts`**

`book-details.component.htm`

```
<div *ngIf="book">
    <p>Questo è book-details.component.html renderizzato separatamente dentro app.component.html al tag <strong>book-detail</strong></p>
    <div class="row selected-book">
        <div class="col-md-4">
            <img [src]="book.coverImage" alt="cover image"/>
        </div>
        <div class="col-md-8">
            <h3 class="title">{{book.title}}</h3>
                <p>{{book.authors}}</p>
                <p>{{book.published}}</p>
                <p>{{book.description}}</p>
        </div>
    </div>
    <div class="row">
        <div class="col-xs-10">
                <button class="btn btn-danger float-xs-right mt-1">
                    Delete
                </button>
        </div>
    </div>
</div>
```

`book.details.component.ts`

```
import { Component, Input } from '@angular/core';
import { Book } from '../model/book';

@Component({
    selector: 'book-details',
    templateUrl: './book-details.component.html'
})

export class BookDetailsComponent {
    @Input() book: Book
}
```

`app.module.ts` in modo espone e dichiara **`BookDetailsComponent`**

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

import { BookDetailsComponent } from './book-details/book-details.component';
import { AppComponent } from './app.component';

@NgModule({
```

```
  declarations: [
    AppComponent,
    BookDetailsComponent
  ],
  imports: [
    BrowserModule,
    FormsModule,
    HttpModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

e infine modificare l'html app.component.html

```
  <!-- lato destro, con il dettaglio del libro -->

     <div class="col-md-7">
          <book-details [book]="selectedBook"> </book-details>
     </div>
```

E aggiungere **`<book-details>`** tag che conterrà book-details.component*

Direttiva @Input

Con questo esempio vediamo che i membri *ts di A non sono direttamente accessibili da B e che per poterli esporre tra componenti è necessario utilizzare la direttiva **`@Input()`**

```
export class BookDetailsComponent {
    @Input() book: Book
}
```

Il binding nell'html viene eseguito attraverso la seguente sintassi

```
         <book-details [book]="selectedBook"> </book-details>
```

Che significa letteralemente "quando selectedBook è diverso da null

```
  getBookDetails (isbn: number) {
        var selectedBooks = this.booksList.filter(book => book.isbn === isbn);
         this.selectedBook = selectedBooks[0];
     }
```

.. assegnalo a book e rivaluta l'*ngIf"

Questo è il modo con cui Angular passa le informazioni tra padri e figli.
Adesso vediamo come restituire le informazioni tra un figlio e il padre

Aliasing delle proprietà

Se volessimo dare un nome diverso alla proprietà del componente B potremmo utilizzare un alias

```
    @Input("myBook") book: Book
```

Nell'html useremo

```
       <book-details [myBook]="selectedBook"> </book-details>
```

Proprietà di Outupt

Vediamo come implementare il pulsante **Delete** attraverso la funzione **`deleteBook()`** (nel padre app.component.ts)

Poi nel ts figlio aggiungiamo il richiamo alla funzione

```
export class BookDetailsComponent {
    @Input("myBook") book: Book
    deleteBook () {  }
}
```

E infine nell'html mettiamo l'evento click al pulsante

```
<button (click)="deleteBook()" class="btn btn-danger float-xs-right mt-1">
    Delete
</button>
```

Abbiamo fatto. Questo codice effettivamente cancella il record ma non rigenera l'output, quindi dobbiamo intercettare l'evento per ottenere la nuova lista dei libri subito dopo il delete. Per farlo ci serve la direttiva **`@Output()`**

```
import { Component, Input, Output, EventEmitter } from '@angular/core';
import { Book } from '../model/book';

@Component({
    selector: 'book-details',
    templateUrl: './book-details.component.html'
})

export class BookDetailsComponent {
    @Input("myBook") book: Book;

    @Output() onDelete = new EventEmitter<number>();

    deleteBook () {
          this.onDelete.emit(this.book.isbn);
    }
}
```

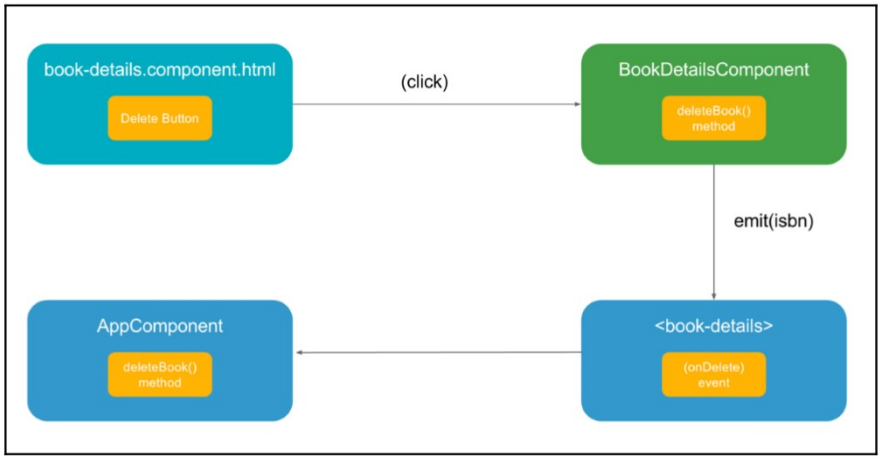
Dobbiamo aggiungere nel ts figlio la direttiva Output ed EventEmitter, dopodichè all'onDelete creare una nuova istanza e nella **`deleteBook()`** dobbiamo inserire le istruzioni riguardanti l'evento

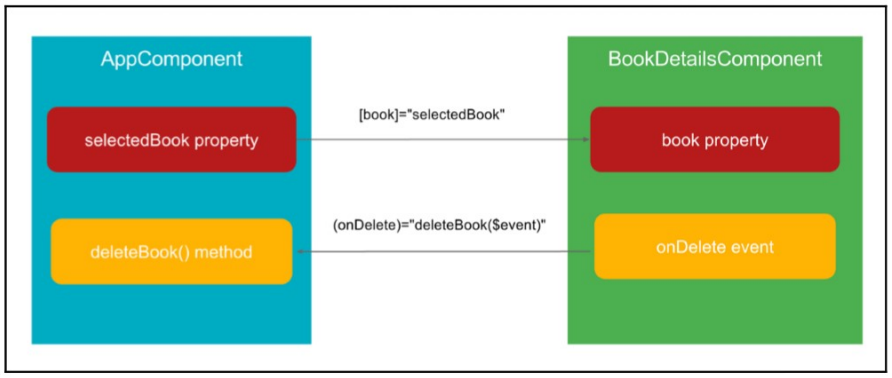
Nel componente padre dobbiamo indicare quale evento rigenera il contenuto (l'onDelete che abbiamo invocato noi)

```
<div class="col-md-7">
    <book-details [myBook]="selectedBook" (onDelete)="deleteBook($event)"> </book-details>
</div>
```

Riepiloghiamo cosa abbiamo fatto finora

- Abbiamo dichiarato un evento onDelete usando la direttiva @Output().
- La proprietà onDelete viene inizializzata come una istanza EventEmitter della classe EventEmitter.
- Sono stati usati oggetti EventEmitter per creare e fare il trigger dell'evento personalizzato
- La proprietà onDelete è un evento, abbiamo usato la sintassi per eseguire il binding tra deleteBook nel componente padre e il deleteBook() nel figlio, ed abbiamo usato un EventEmitter e un metodo emit() passandogli l'ISBN del libro visualizzato al momento.
- Quando un evento onDelete è triggerato nel figlio, invoca il metodo deleteBook() nell'AppComponent
- Abbiamo passato il numero ISBN usando l'oggetto $event nel deleteBook di AppComponent.
- **Abbiamo creato un oggetto EventEmitter usando il tipo di dati number perche abbiamo passato all'evento un parametro di questo tipo. Possiamo usare qualsiasi altro tipo di dati**





Condivisione dati via servizi

Nell'esempio precedente abbiamo implementato l'app attraverso una struttura di dati mockati, ovvero tre libri preimpostati, oggetti precaricati a runtime per il debug, test, demo.
In una applicazione reale avremo la necessita di accedere ai dati su fonti esterne via servizi Rest. Lo scopo è quello di avere il contesto accessibile in qualsiasi punto dell'app, su piu componenti. Per farlo dobbiamo implementare un singolo e riusabile punto di accesso attraverso un servizio Angular. Un servizio Angular scritto in TypeScript è una semplice classe che funge da access point riutilizzabile.
Dobbiamo fare un po di refactoring alla nostra app dei libri.
Fai un backup e salvala con un nome diverso e cambia nome a questa chiamandola
`Master-book2-service`
I files su cui cambiare nome sono e2e\app.e2e-spec.ts, package.json, .angular-cli.json
Riscriviamo la logica del nostro codice per accedere al servizio e catturare i dati, cancellarli, aggiungerli.

Creiamo il file **`app/book-store.service.ts`**

```
import { Injectable } from '@angular/core';
import { Book } from './model/book';
import { BOOKS } from './model/mock-books';

@Injectable() export class BookStoreService {
  booksList: Book[] = BOOKS;

  getBooks () {    return this.booksList;  }

  getBook (isbn: number) {
```

```
      var selectedBook = this.booksList.filter(book => book.isbn === isbn);
      return selectedBook[0];
    }

  deleteBook (isbn: number) {
      this.booksList = this.booksList.filter(book => book.isbn !== isbn);
      return this.booksList;  }
  }
```

In una app reale i metodi esposti in questo ts potrebbero comunicare con servizi rest esterni usanto i meccanismo tipili XHR e JSONP
La logica puo cambiare in ogni momento senza interferire con i componenti esistenti che gia consumano il servizio.
Qua non ci sono elementi degni di nota eccetto il decoratore @Injectable(). Questo decoratore è usato da TypeScript per emettere il metadata del servizio, metadata che angular implementa per 'iniettare' ulteriori dipendenze al nostro servizio. Il nostro BookStoreService non ha nessuna dipendenza al momento, ma stiamo appunto aggiungendo il decoratore per sperimentare e far pratica sulla consistenza dello strumento e sarà molto utile in futuro.

Quindi dobbiamo rivedere il codice nel nostro AppComponent che usi il metodo BookStoreService. Per prima cosa dobbiamo creare l'oggetto BookStoreService. Ne piu ne meno delle altre classi, dobbiamo crearlo con il suo costruttore new.

Modifichiamo il nostro **`app.component.ts`**

```
import { Component, OnInit } from '@angular/core';
import { Book } from './model/book';
import { BOOKS } from './model/mock-books'
import {BookStoreService} from './book-store.service'

@Component({
  selector: 'books-list',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css'],
  providers: [BookStoreService]
})


export class AppComponent implements OnInit {
  booksList: Book[];  selectedBook: Book;
  constructor(private bookStoreService: BookStoreService) { }
  ngOnInit() {
    this.getBooksList();
  }
  getBooksList() {
     this.booksList = this.bookStoreService.getBooks();
    }
  getBookDetails(isbn: number) {
     this.selectedBook = this.bookStoreService.getBook(isbn);
    }
  deleteBook(isbn: number) {
    this.selectedBook = null;
    this.booksList = this.bookStoreService.deleteBook(isbn);
  }
}
```

Vediamo le cose interessanti e cosa si deve sapere

- Abbiamo aggiunto come import del core angular l'oggetto OnInit (Component e OnInit sono due nomi di riferimento allo stesso engine)
- Nel @Component abbiamo introdotto un nuovo elemento, Provider, che sara visibile anche a tutti i componenti figli
- In tutti i figli non c'è bisogno di ridichiarare il provider nel @Component
- Se un figlio dichiara un servizio nella sua array de providers con lo stesso nome dichiarato nel padre, Angular crea una nuova istanza per il figlio e non userà il provider padre

- Un provider è visibile a se stesso e a tutti i suoi figli
- Invece di dichiarare il servizio nell'array dei providers del component, possiamo dichiararlo anche a livello di modulo, nell'array del modulo `@NgModule({ providers: ["xxxxx"] })`
- Un servizio dichiarato a livello di modulo è visibile a tutti i suoi componenti figli e al modulo stesso

Altra cosa importante nella class AppComponente che invece di chiamare il metodo `getBookList()` direttamente nel costruttore lo stiamo chiamanto nel metodo speciale `ngOnInit()`.
`ngOnInit()` è un motodo delegato di hook invocato quando viene creato il componente, ancor prima di raggiungere qualsiasi dato.

Capitolo 3 – Esempio: Server REST per ToDo

In questo capitolo implementeremo un piccolo Server REST API per fornire dati attraverso richieste JSON e una applicazione Client che consuma i dati.

In tutti gli esempi che ho trovato la stessa app faceva sia da Client che da Server. Non c'è nulla di sbagliato, in quanto i metadata sono condivisi e i servizi sono esposti su porte http diverse, ma per un principiante è sicuramente piu facile fare confusione, quindi ho diviso in due app distine, una per il server e una per il client.

Partiamo dal server.

Json-server package

Perche una webapp Angular funga da sever API servono essenzialemente 3 elementi:

- installare il package **json-server**
- preparare un file json contenente i dati
- dire all'app dove prenderli

crea una nuova app

```
ng new server-todo
```

Installa il package

```
npm install json-server --save
```

Crea un set di dati fissi per fare un pò di test (mocked). Aggiungi il file db.json nella root, dove sta il package.json

```
{
   "todos": [
     {
       "id": 1,
       "title": "Leggere la giuda di Angular 5",
       "complete": false
     },
     {
       "id": 2,
       "title": "Ripulire dischi ",
       "complete": false
     },
     {
       "id": 3,
       "title": "Prenota la cena al ristorante per venerdi",
       "complete": false
     }
   ]
}
```

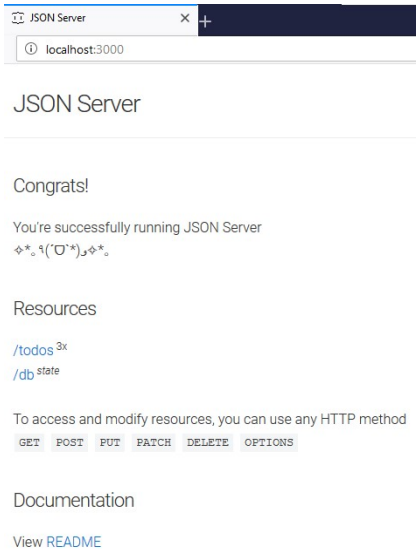
Aggiorna il packege.json. con questo l'app sa in quale file json deve andare a pescare i dati

```
"scripts": {
  ...
  "json-server": "json-server --watch db.json"
}
```

A questo punto puo gia essere mandato in esecuzione, lancia il comando:

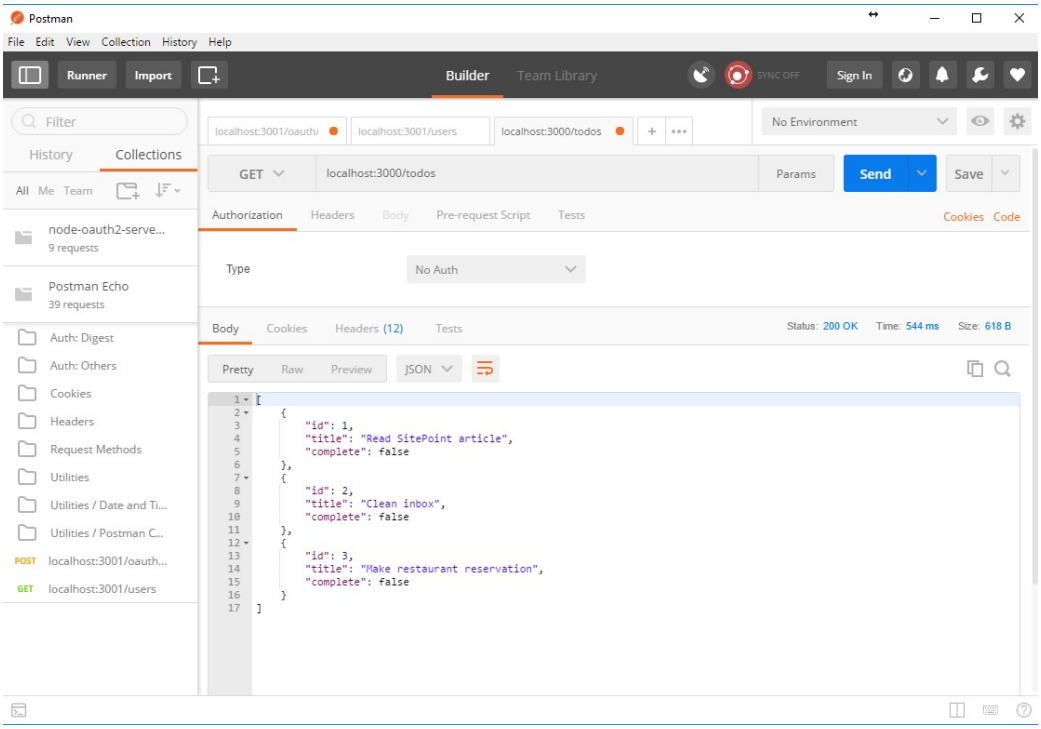
```
npm run json-server
```

Aprendo il browser Il risultato deve essere questo

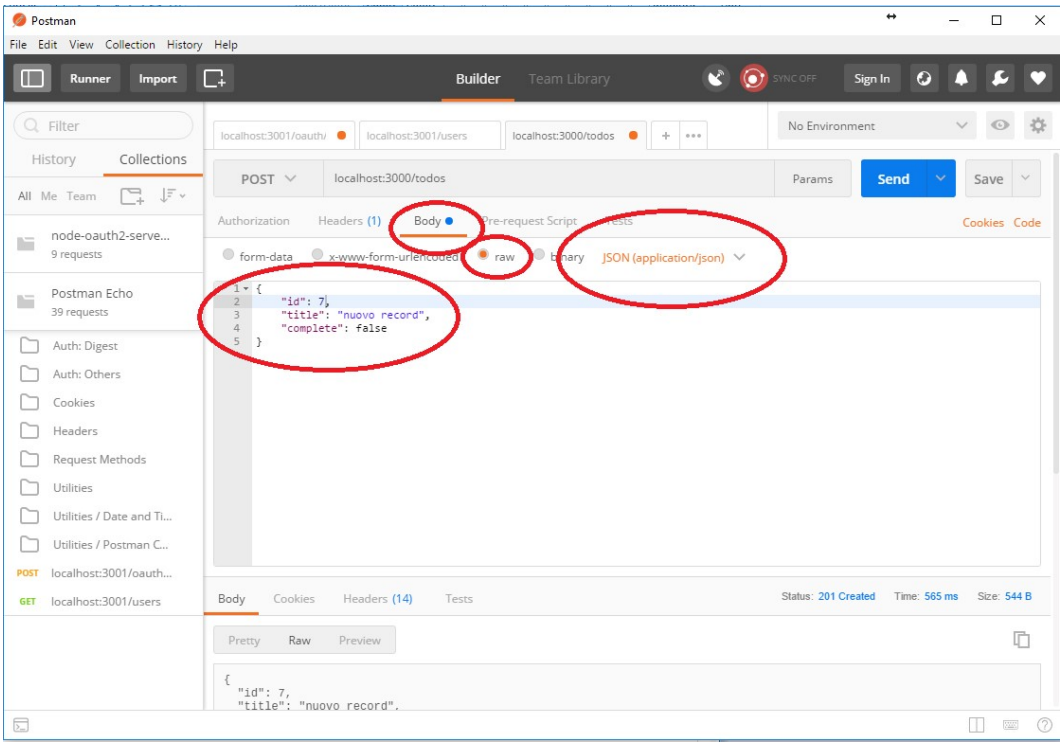


Postman

Per provare tutte le funzionalità del server basta utilizzare un client http, il piu famoso è PostMan
https://www.getpostman.com/apps



Se vuoi aggiungere un nuovo record esegui un POST con un nuovo record JSON



Con lo stesso metodo usa il PUT per aggiornare un esistente

Il DELETE per cancellare un record

Ed ecco fatto il nostro server JSON. Il nostro serverino accetta tutti i comandi di gestione e li salva fisicamente su disco. Il nuovo file json contente i dati, se vedi da Visual Studio Code, contiene il nostro record con id 7


```
{
  "todos": [
    {
      "id": 1,
      "title": "Read SitePoint article",
      "complete": false
    },
    {
      "id": 2,
      "title": "Clean inbox",
      "complete": false
    },
    {
      "id": 3,
      "title": "Make restaurant reservation",
      "complete": false
    },
    {
      "id": 7,
      "title": "nuovo record",
      "complete": false
    }
  ]
}
```


Il server rest Json che abbiamo appena visto possiamo controllare un database su json, e anche a intuito un database json non è il massimo del realismo. In ambienti di produzione avremo API che espongono dati prendendo come sorgente o provider un SQL Server, mySql, noSQL, Oracle, MongoDB o uno delle piattaforme professionali

Client REST

Abbiamo il server che espone i nostri dati, costruiamo un client consumer dell'endpoint appena implementato

Fai la copia della app ToDo vista nel capitolo 1 e chiamala Client-Rest-ToDo nel **"name"** del file .angular-cli.json

Ora aggiungiamo il supporto per i seguenti endpoint, la base url come abbiamo visto è **http://localhost:3000**

```
    GET /todos: get all existing todo's
    GET /todos/:id: get an existing todo
    POST /todos: create a new todo
    PUT /todos/:id: update an existing todo
    DELETE /todos/:id: delete an existing todo
```

Metti nell'environment una variabile che contenga il base Url, cosi se cambia in futuro non è hardcodato in giro per i sorgenti: nel file **envinonment.ts**

```
export const environment = {
  production: false,
  // URL of development API
  apiUrl: 'http://localhost:3000'
};
```

Creazione API Service per la comunicazione

Esegui il comando

```
ng generate service Api --module app.module.ts
```

Se andiamo a vedere, l'opzione **–module** ha detto al CLI non solo di creare l'**api.service.ts** ma anche di registrarlo nell'**app.module.ts**

Nell'**api.service.ts** iniettaci l'environment e l' http nativo di Angular

```
import { Injectable } from '@angular/core';
import { environment } from 'environments/environment';
import { Http } from '@angular/http';

const API_URL = environment.apiUrl;

@Injectable()
export class ApiService {
  constructor(  private http: Http  ) {     }
}
```

Prima di procedere approfondiamo il servizio http

HTTP Service di Angular

Il servizio è disponibile come classe http attraverso l'import

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

Questo corrisponde al nostro Client http che incapsula funzioni native XHR/JSONP :

```
    delete(url, options): esegue una richiesta DELETE
    get(url, options): esegue una richiesta GET
    head(url, options): esegue una richiesta HEAD
    options(url, options): esegue una richiesta OPTIONS
    patch(url, body, options): esegue una richiesta PATCH
    post(url, body, options): esegue una richiesta POST
    put(url, body, options): esegue una richiesta PUT
```

Ciascuno di questi metodi restituisce un oggetto **RxJS** Observable. A differenza di AngularJS 1 che restituiva Promises, dalla 2 in poi il tipo di oggetto restituito è sempre un Observable

Nel prossimo capitolo vedremo piu in dettaglio RxJS, per il momento limitiamoci a trattare questi oggetti come esempio. Definiamo i membri pubblici di gestione **getAllTodos(), createTodo(), updateTodo(), DeleteTodoById()**

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

import { Http, Response } from '@angular/http';
import { TodoItem } from './Model';
import { Observable } from 'rxjs/Observable';

const API_URL = environment.apiUrl;

@Injectable()
export class ApiService {

  constructor(    private http: Http   ) {  }

  // API: GET /todos
  public getAllTodos() {
    // userà this.http.get()
  }

  // API: POST /todos
  public createTodo(todo: TodoItem) {
    // userà this.http.post()
  }

  // API: GET /todos/:id
  public getTodoById(todoId: number) {
    // userà  this.http.get()
  }

  // API: PUT /todos/:id
  public updateTodo(todo: TodoItem) {
    // userà this.http.put()
  }

  // DELETE /todos/:id
  public deleteTodoById(todoId: number) {
    // userà this.http.delete()
  }
}
```

Vediamo che il **TodoItem** è lo stesso tipo di dati che usavamo nell'app ToDo con i dati mockati internamente

Implementazione Metodi Api.Service

Implentiamo ora le chiamate alle Api del server precedente

```
GET /todos: lista completa di tutti i todo
GET /todos/:id: aggancio di una singola scheda ToDo
POST /todos: crea un nuovo ToDo
PUT /todos/:id: Aggiorna un toDo esistente
DELETE /todos/:id: Cancella una scheda ToDo
```

GetAllTodos()

Con questo metodo vogliamo la lista completa di tutti gli elementi contenuti nella collection di tipo TodoItem

```
  // API: GET /todos
public getAllTodos(): Observable<TodoItem[]> {
  return this.http
    .get(API_URL + '/todos')
    .map(response => {
      const todos = response.json();
      return todos.map((todo) => new TodoItem(todo,false));
    });
}
```

Come prima cosa facciamo una richiesta GET all'endpoint passato come parametro

```
    this.http.get(API_URL + '/todos')
```

che restituisce un **Observable<Response>**

Dopodichè invochiamo il metodo RxJS **map** per trasformare l'observable in una array di **ToDoItem**

```
.map(response => {
        const todos = response.json();
        return todos.map((todo) => new TodoItem(todo, false));
      })
```

Il map dichiara localmente todos a cui assegna il json presente nel response e restituisce, per ogni elemento todo di tipo ToDoItem un nuovo oggetto nell'array.
Perché tutto funzioni dobbiamo importare gli operatori corrispondenti

```
import "rxjs/add/operator/map";
import 'rxjs/add/operator/catch';
import 'rxjs/add/observable/throw';
```

Infine completiamo gli altri metodi. Spostiamoci sul data model
Partiamo da una classe sincrona in cui i dati vengono letti e salvati in memoria (**`model.ts`**)

```
export class Model {
    user;
    items;
    lastId: number = 0;
    constructor() {
        this.user = "Adam";
        this.items = [new TodoItem("Buy Flowers", false),
                      new TodoItem("Get Shoes", false),
                      new TodoItem("Collect Tickets", false),
                      new TodoItem("Call Joe", false)]

    }
      // Simulate POST /todos
  addTodo(todo: TodoItem): Model {
    if (!todo.id) {
      todo.id = ++this.lastId;
    }
    this.items.push(todo);
    return this;
  }

  complete()
  {
    //this.items.map(x=>x.id)
  }

// Simulate DELETE /todos/:id
deleteTodoById(id: number): Model {
    this.items = this.items
                  .filter(todo => todo.id !== id);
    return this;
  }

  // Simulate PUT /todos/:id
  updateTodoById(id: number, values: Object = {}): TodoItem {
    let todo = this.getTodoById(id);
    if (!todo) {
      return null;
    }
    Object.assign(todo, values);
    return todo;
  }

  // Simulate GET /todos
  getAllTodos(): TodoItem[] {
    return this.items;
  }

  // Simulate GET /todos/:id
  getTodoById(id: number): TodoItem {
    return this.items
      .filter(todo => todo.id === id)
      .pop();
  }
}

export class TodoItem {
    id;
    action;
    done;

    constructor(action, done) {
        this.action = action;
        this.done = done;
    }

}
```

Aggiungiamo il nostro nuovo servizio asincrono che interroga l'API nel servizio api.service
File: **`model.service.ts`**

```
import { Injectable } from '@angular/core';
import { ApiService } from './api.service';
import { TodoItem } from './model';
import { Observable } from 'rxjs/Observable';

@Injectable()
export class TodoService {

    constructor(private api:ApiService) {

    }
      // Simulate POST /todos
  addTodo(todo: TodoItem): Observable<TodoItem> {
    return this.api.createTodo(todo);
  }

  complete()
  {
    //this.items.map(x=>x.id)
```

```
  }

// Simulate DELETE /todos/:id
  deleteTodoById(id: number): Observable<TodoItem> {
    return this.api.deleteTodoById(id);
  }

  // Simulate PUT /todos/:id
  updateTodo(todo: TodoItem): Observable<TodoItem>  {
    return this.api.updateTodo(todo);
  }

  // Simulate GET /todos
  getAllTodos(): Observable<TodoItem[]> {
    return this.api.getAllTodos();
  }

  // Simulate GET /todos/:id
  getTodoById(id: number): Observable<TodoItem>  {
    return this.api.getTodoById(id);
  }
}
```

Le differenze sono facili da intuire. Tutto il lavoro è demandato all'api, nel ToDoService vengono solo richiamati i metodi che ci servono e il lavoro sui dati lo fa il modulo del servizio API.
In questo esempio abbiamo visto applicati i primi concetti Observable e di Programmazione reattiva che esploreremo nel prossimo capitolo

Capitolo 4 - RxJS

Lavorare con gli Observables

In questo capitolo esploriamo il *paradigma di programmazione reattiva* ed analizzaremo il flusso di dati verso una applicazione. Usiamo oggetti **Observables** per implementare i concetti di programmazione reattiva. L'ES7 ha la prerogativa di includere oggetti Observables all'interno del linguaggio JavaScript. Oggigiorno possiamo usarli con le librerie di estensioni reattive (Reactive-Extension for Javascript - **RxJS**). Questo capitolo coprirà solo i concetti essenziali dell'RxJS e nello specifico analizzeremo

- Basi di Programmazione Reattiva RxJS
- Cosa sono gli oggetti Observables e i suoi operatori
- Scrittura di componenti e servizi Observables

Cosa è RxJS

Rx è una libreria supportata da diversi linguaggi di programmazione ed implementano gli stessi concetti esposti di seguito. I linguaggi più diffusi sono:

- Java: RxJava
- JavaScript: **RxJS**
- C#: Rx.NET
- C#(Unity): UniRx
- Scala: RxScala
- Clojure: RxClojure
- C++: RxCpp
- Lua: RxLua
- Ruby: Rx.rb
- Python: RxPY
- Go: RxGo
- Groovy: RxGroovy
- JRuby: RxJRuby
- Kotlin: RxKotlin
- Swift: RxSwift
- PHP: RxPHP
- Elixir: reaxive
- Dart: RxDart

Programmazione Reattiva

Nella programmazione tradizionale lo stato di una variabile cambia quando c'è una assegnazione esplicita di un nuovo valore. In questo caso la variabile perde il valore precedente; qui il dato è propagato usando un meccanismo di pull, qualsiasi dipendenza da questa variabile o oggetto deve fare il pull esplicito del valore quando ci sono nuove modifiche. Non avvengono propagazioni automatiche.
La programmazione reattiva lavora in modo opposto. Invece di una assegnazione esplicita effettua un push implicito e le modifiche vengono propagate automaticamente a tutte le dipendenze. In altre parole 'reattiva' sta appunto l'invocazione di specifici triggers sulle varabili o oggetti al loro cambio di stato.

Angular adotta questo questo nuovo paradigma di programmazione che è destinato a cambiare il modo di sviluppare facendo astrarre moltissimi concetti ora di fatto statici.

Dati asincroni

Per estendere ancora il concetto, Reactive Programming vuol dire programmare facendo uso di flussi di dati asincroni. In uno scenario che non è nuovo ai Front-End Developer i click in una pagina web rappresentano eventi oppure , nel mondo Reactive, un flusso asincrono in cui l'evento insiste e che può essere osservato al fine di reagire con azioni determinate.

Puoi creare flussi di dati (data streams) di qualsiasi cosa: variabili, input utente, proprietà, cache, strutture..ecc. Puoi quindi osservare cosa accade e reagire in modo coerente.

I flussi (streams) sono centrali nelle tecnologie Reactive come RxJS perché possono essere, oltre che creati, combinati tra di loro e filtrati. In buona sostanza un flusso può usato come input per un altro flusso, flussi multipli posso essere combinati come input oppure può essere effettuato un merge. Inoltre ai flussi possono essere applicati filtri al fine, magari, di recuperare solo gli eventi che ci interessano ed infine i valori di un flusso possono essere mappati su un altro flusso.

Observer

L'**`Observer`** è una collection di callbacks che stanno in ascolto sul valore emesso da un Observable: La sua interfaccia (non indispensabile in Angular ma è per capire la struttura dei callback)

```
interface Observer<T> {
     closed?: boolean;
     next: (value: T) => void;
     error: (err: any) => void;
     complete: () => void;
    }
```

L'oggetto Observer è costituito da un albero di metodi: **`next(), error() e complete().`**
Questi metodi sono spiegati nel dettaglio

- Tutte le volte che un Observable emette (emit) il valore viene invocato il callback **`next()`**
- Se non ci sono altri valori emessi dall'Observabe viene invocato il callback **`complete()`**
- Il callback **`error()`** se viene sollevato un errore. L'Observer ferma l'ascolto del valore sulla variabile

Observable

L'**Observable** è una collection di valori o di eventi; può essere un modello di eventi, una richiesta server asincrona, una animazione UI. La classe Observable ha diversi metodi per creare collections di Observable

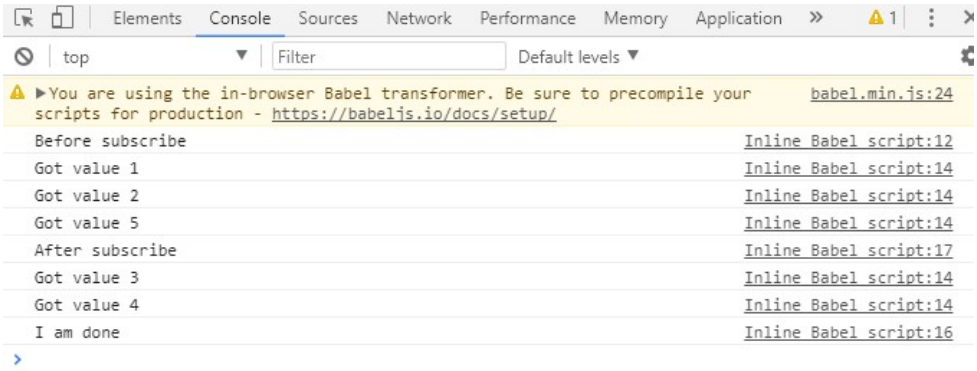
`Observable.create()`
`Observable.of()`
`Observable.from()`
`Observable.fromArray()`
`Observable.fromEvent()`
`Observable.fromPromise()`
`Observable.interval()`
`Observable.timer()`

L'`Observable` costituisce il core dei principi **RxJS** ; è molto importante capire bene come usarli e come li usa Angular al suo interno. Iniziamo con utilizzare il metodo `Observable.create()` per generarlo:

Per vederlo in azione creiamo un file `observable1.html`

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="UTF-8">
        <title>Manually creating an Observable</title>
    </head>
    <body>
    <script type="text/babel">
        const observable = Rx.Observable.create((observer) => {
            observer.next(1);
            observer.next(2);
            setTimeout(() => {
                observer.next(3);
                observer.next(4);
                observer.complete();
            }, 1000);
            observer.next(5);
        });
        console.log('Before subscribe');
        observable.subscribe({
            next: val => console.log(`Got value ${val}`),
            error: err => console.log(`Something went wrong ${err}`),
            complete: () => console.log('I am done')  });
        console.log('After subscribe');
    </script>
    <script src="https://unpkg.com/babel-standalone@6/babel.min.js"> </script>
    <script src="https://unpkg.com/@reactivex/rxjs/dist/global/Rx.js"> </script>

    </body>
</html>
```

Se lo apriamo nel browser possiamo osservere in Console il risultato seguente:



Il metodo `.create()` crea un nuovo Observable usando un Observer. L'Observable emetterà il valore solo quando un Observer si registra usando `.subscribe()`. L'output in console rispetta quanto detto finora. Appena capirò un effettivo utilizzo pratico lo spiego.

Babel

Nel codice precedente ha fatto la sua prima apparizione **babel**

```
    <script type="text/babel">
```

Babel https://babeljs.io/ è un compilatore JavaScript, è già incluso in NodeJS e il suo scopo è di tradurre Javascript ES6 in formato vecchio stile, quello che si vedeva nel primo decennio del 2000.
Se ad esempio hai un tuo vecchio ma stabile e funzionante script in javascript e lo vuoi usare devi dichiararlo come script babel, altrimenti non sa cosa è perche si aspetta TypeScript.
Qui sopra infatti non vediamo JavaScript antico, ma qualcosa che sembra TypeScript.

Infatti babel ha valenza bidirezionale, puoi scrivere in vecchissimo JavaScript e viene intepretato, oppure scrivere superset del nuovo Javascript ES6 e i vecchissimi browser su computer vecchissimi apriranno regolarmente tutte le tue pagine Angular o NodeJS
Note bene che babel assomiglia a TypeScript perche di per se TS è sintassi babel, ma TS ha un numero maggiore di istruzioni che babel non sa cosa siano.

TypeScript is both a ES6 to ES3+ transpiler and a superset of javascript that it adds somewhat capabilities to javascript,whereas Bable only transpiles ES6 to ES5

Proseguiamo con gli Observables.
Di seguito un esempio che agisce sul DOM usando un Observable. Queste due linee loggano il movimento del mouse sulla console del browser:

```
        const mouseMoves = Rx.Observable.fromEvent(document, 'mousemove');
        mouseMoves.subscribe(event => console.log(event.clientX, event.clientY));
```

Subscription

L'oggetto **`subscription`** rappresenta l'esecuzione di un Observable ed è utilizzato anche per cancellare l'esecuzione. Il codice **`esempio3.html`**:

```
const interval = Rx.Observable.interval(1000);
const subscription = interval.subscribe(val => console.log(val));
setTimeout(() => {  subscription.unsubscribe(); }, 10000);
```

In questo esempio l'Observable emette un valore ogni secondo e lo logghiamo sulla console. Dopodichè stoppa l'emissione dopo 10 secondi attraverso `l'`**`unsubscribe().`**

Operatori

Un operatore è una funzione pura che crea un nuovo Observable basato sull'Observable corrente e ci permette di effettuare diversi tipi di operazioni come il *filtering, mapping e ritardo dei valori*. RxJS è ricchissimo di operatori e ne vedremo un bel po nel corso di questa guida.

Di seguito un esempio di operatori **`map()`** e **`filter()`** sul codice di esempio4.html. Usiamo map() per moltiplicare i valori, e quindi usiamo **`filter()`** per filtrare i valori dispari.

```
const interval = Rx.Observable.interval(1000).map(x => x * 2).filter(x => x%2 === 0);
interval.subscribe(val => console.log(val))
```

Esempi: Salva il codice seguente in una pagina es:. **`observable.html`**, aprila con Chrome e premi F12, segui l'output sulla console

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="UTF-8">
        <title>Manually creating an Observable</title>
    </head>
    <body>
    <script type="text/babel">
// esempio1.html
        const observable = Rx.Observable.create((observer) => {
            observer.next(1);
            observer.next(2);
            setTimeout(() => {
```

```
                observer.next(3);
                observer.next(4);
                observer.complete();
            }, 1000);
            observer.next(5);
        });
        console.log('Before subscribe');
        observable.subscribe({
            next: val => console.log(`Got value ${val}`),
            error: err => console.log(`Something went wrong ${err}`),
            complete: () => console.log('I am done')  });
        console.log('After subscribe');
// esempio2.html
        const mouseMoves = Rx.Observable.fromEvent(document, 'mousemove');
        mouseMoves.subscribe(event => console.log(event.clientX, event.clientY));
// esempio3.html
        const interval = Rx.Observable.interval(1000);
        const subscription = interval.subscribe(val => console.log(val));
        setTimeout(() => {  subscription.unsubscribe(); }, 10000);

// esempio4.html
        const interval1 = Rx.Observable.interval(1000).map(x => x * 2).filter(x => x%2 === 0);
        interval1.subscribe(val => console.log(val))

    </script>


    <script src="https://unpkg.com/babel-standalone@6/babel.min.js"> </script>
    <script src="https://unpkg.com/@reactivex/rxjs/dist/global/Rx.js"> </script>

    </body>
</html>
```

Come si puo notare questa è una semplice pagina html, non occorre avviare motori server, NodeJS ecc, babel è direttamente interpretato dal browser (dopo aver caricato lo script **`babel.mini.js`**)

Riepilogo Operatori più comuni

Operatore	Descrizione	Esempio
filter	Filtra i valori restituiti da un Observable in base ad una condizione booleana specificata	filter(x => x % 2 == 0)
interval	Restituisce un Observable che genera un valore numerico ad ogni intervallo di tempo prefissato espresso in millisecondi	interval(1000)
last	Restituisce un Observable che contiene l'ultimo valore generato dall'Observable originario	last()
map	Restituisce un Oblervable che moltiplica valori	map(x=> x * 2)
max	Restituisce il valore massimo generato da un Observable	max()
merge	Combina più Observable creandone uno che restituisce i valori asincroni generati da ciascuno di essi	mergedObservable = observ1.merge(observ2, observ3);
concat	Restituisce un Observable che concatena i valori restituiti da due o più Observable	concatObservable = observ1.concat(observ2, observ3);
retry	Restituisce un Observable identico all'originale sul quale, in caso di errore, si autoregistra per rieseguire l'operazione asincrona fino ad un massimo stabilito dal parametro passato	retry(3)
fromEvent	Restituisce un Observable a seguito di un evento sollevato sul DOM	fromEvent(document,'mousemove')
create	Crea un istanza Observable	Create((observer) => {..})

Per il dettaglio di ogni operatore ed esempi vedere il link
https://github.com/Reactive-Extensions/RxJS/blob/master/doc/api/core/observable.md

Altri esempi

A fronte dell'interfaccia prototipo

```
Observer<String> traveller = new Observer<String>() {
        onCompleted() {         console.log("My trip is finished");          }
        onError(Throwable e) {  console.log("I won't complete my trip!");    }
        onNext(String t) {      console.log ("I've just visited " + t);      }
    }
```

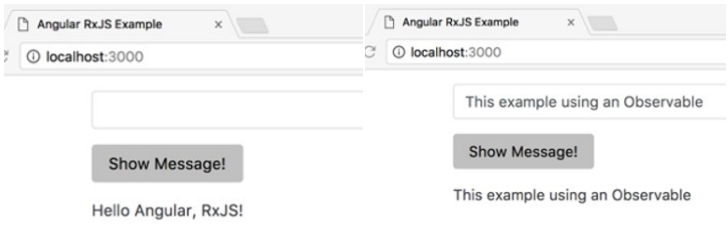
Merging Observable streams

Nell'esempio precedente abbiamo due blocchi subscribe ridondanti che fanno la stessa cosa. Possiamo fare un refactoring utilizzando l'operatore **`merge().`**

Modifichiamo la parte **`ngOnInit()`** e Includiamo il **`merge()`** importandolo da **`'rxjs/add/operator/merge'`**:

```
import 'rxjs/add/operator/merge';
    ..
    ngOnInit() {
        const btnOb$ = Observable
                        .fromEvent(this.btn.nativeElement, 'click')
                        .map(event => 'Hello Angular, RxJS!');
        const textOb$ = Observable
                        .fromEvent(this.text.nativeElement, 'change')
                        .map(event => event.target.value);
        Observable
        .merge(btnOb$, textOb$)
        .subscribe(res => this.message = res);
        }
    }
    ..
```

Usiamo il **`merge()`** per combinare sia lo stream che la subscribing sull'output, e contestualmente emette tutti i valori per ogni input Observable. Nel nostro caso entrambi, il click dell'utente o l'inserimento del testo nel textbox produce la visualizzazione del messaggio



Observable.interval()

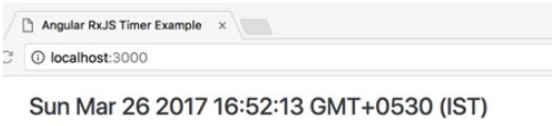
```
import { Component, OnInit } from '@angular/core';
import { Observable } from 'rxjs/Observable';
import 'rxjs/add/observable/interval';
import 'rxjs/add/operator/map';

@Component({
    selector: 'app-root',
    template: `<div class="container">
                    <p class="mt-1">{{time}}</p>
                </div>`}
)

export class AppComponent implements OnInit {
    time: string;
    ngOnInit() {
            const timer$ = Observable.interval(1000).map(event => new Date());
            timer$.subscribe(val => this.time = val.toString());
       }
}
```

Questo aggiorna un timer e visualizza l'ora ogni secondo.

Adesso lo modifichiamo affinche invece di fare un subscribing dell'Observable timer$, lo visualizziamo direttamente sulla pagina:

```
export class AppComponent {
    timer$ = Observable.interval(1000).map(event => new Date());
}
```

Abbiamo mostrato entrambi gli approcci per evidenziare meglio le differenze e su cosa agiscono gli operatori e i metodi



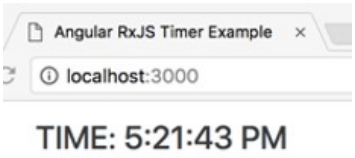
AsyncPipe

Questa tecnica consiste nel fare la subscribe di un Observable internamente e fare in modo che restituisca l'ultimo valore che ha emesso.

```
  template: `<div class="container">    <h4 class="mt-1">{{timer$ | async}}</h4>  </div>'
```

E adesso prendiamo lo stesso output di prima ma senza la subscribing diretta all'Observable. Formattiamo anche la data usando il **`DatePipe`** per visualizzare solo l'ora ('mediumTime'):

```
template: `<div class="container">    <h4 class="mt-1">    TIME: {{timer$ | async | date: 'mediumTime'}}    </h4>  </div>'
```



HTTP e Observable

Implementare il campo di ricerca sull'app Book

Per capire gli Observable in dettaglio torniamo al nostro progetto precedente per la gestione dei Libri. Copiamo la cartella **`master-book2`** in **`master-book3`** e modifichiamo il nome app nei 3 file menzionati in precedenza. A questo progetto aggiungeremo la funzionalità di ricerca attraverso Observables e Service providers, tutti concetti che abbiamo esposto fino a qui le integriamo nella nostra app.

Abbiamo bisogno di un form di ricerca, dalla root

`ng generate component book-search`

`book-search.component.ts`

```
import { Component, OnInit, ViewChild } from '@angular/core';
import { Observable } from 'rxjs/Observable';
import {BookStoreService} from '../book-store.service'
// operatori rxjs
import 'rxjs/add/observable/fromEvent';
import 'rxjs/add/operator/map';
import "rxjs/add/operator/debounceTime";
import "rxjs/add/operator/distinctUntilChanged";
import "rxjs/add/operator/switchMap";


@Component({
  moduleId: module.id,
  selector: 'book-search',
  styleUrls: ['./book-search.component.css'],
  template: `    <h3 class="page-title">Books Search</h3>
                <div class="search-container">
                <div class="books-search-form">
                <input type="text" #searchInput  class="search-input" placeholder="Book Title">
                <button class="btn btn-primary">Search</button>
                </div>
                <ul>
                <li *ngFor="let bookTitle of bookTitles">        {{bookTitle}}  </li>
                </ul>
                </div> `
  })
```

```
export class BookSearchComponent implements OnInit {
   @ViewChild('searchInput') searchInput;
   bookTitles: Array<string>;


   constructor(private bookStoreService: BookStoreService) { }

    ngOnInit() {
      Observable.fromEvent(this.searchInput.nativeElement, 'keyup')
      .debounceTime(400)
      .distinctUntilChanged()
      .map((event: KeyboardEvent) =>
      (<HTMLInputElement>event.target).value)
      .switchMap(title =>
        this.bookStoreService.getBookTitles(title))
        .subscribe(bookTitles => this.bookTitles = bookTitles);
   }
  }
}
```

L'operatore **`debounceTime (400)`** attende 400 ms dopo ogni input da tastiera prima elaborare la ricerca, l'operatore **`distinctUntilChanged()`** lo ignora se il termine di ricerca successivo è uguale a quello precedente. Ora stiamo utilizzando l'operatore **`switchMap ()`** anziché l'operatore **`mergeMap ();`** si passa ad un nuovo Observable ogni volta che il termine di ricerca cambia.

Effettuiamo ulteriori modifiche al nostro codice per inviare il termine di ricerca al componente padre

Lasciamo **`book-search.component.html`** e **`book-search.component.css`** vuoti

`app.component.ts`

```
import { Component, OnInit } from '@angular/core';
import { Book } from './model/book';
import { BOOKS } from './model/mock-books'
import {BookStoreService} from './book-store.service'

@Component({
  selector: 'books-list',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css'],
  providers: [BookStoreService]

})

export class AppComponent implements OnInit {

  constructor(private bookStoreService: BookStoreService) { }
  title = 'Elenco Libri';
  booksList: Book[] = BOOKS;
  selectedBook: Book;

  ngOnInit() {
    this.getBooksList();
  }
  getBooksList() {
     this.booksList = this.bookStoreService.getBooksAll();
    }
  getBookDetails(isbn: number) {
     this.selectedBook = this.bookStoreService.getBook(isbn);
    }
  deleteBook(isbn: number) {
    this.selectedBook = null;
    this.booksList = this.bookStoreService.deleteBook(isbn);
  }
}
```

Dopodiche apri il service **`book-store.service.ts`** e modificalo

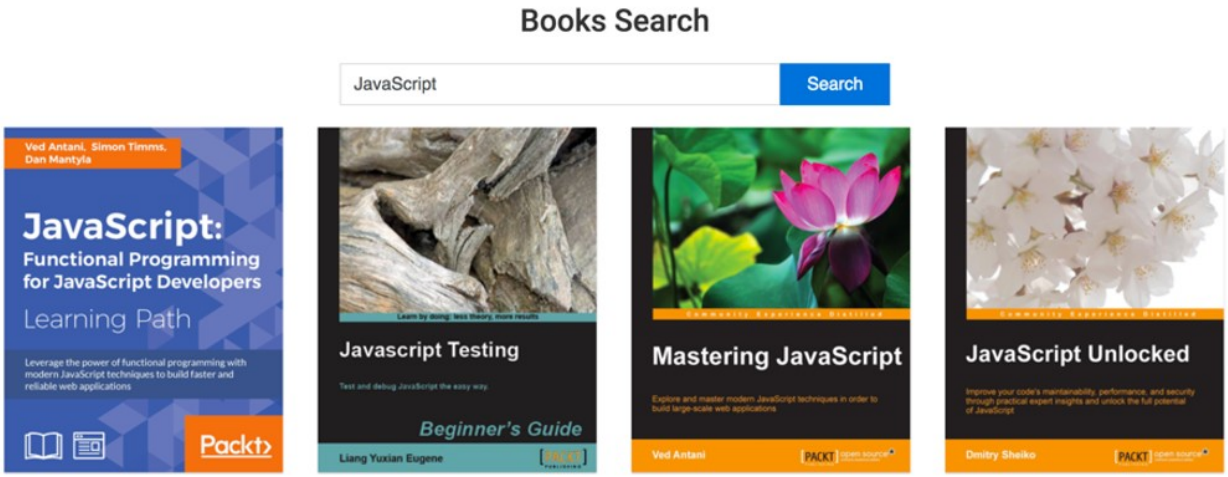
```
import { Injectable } from '@angular/core';
import { Observable } from 'rxjs/Observable';
import 'rxjs/add/observable/of';
import { Book } from './model/book';
import { BOOKS } from './model/mock-books';


@Injectable()
export class BookStoreService {


    booksList: Book[] = BOOKS;

    // metodi

    getBooksAll () {    return this.booksList;  }

    getBooks(title: string): Observable<Book[]> {    return Observable.of(this.filterBooks(title));  }
```

```
    getBookTitles(title: string): Observable<string[]> {
        return Observable.of(this.filterBooks(title).map(book => book.title));
    }

    filterBooks(title: string): Book[] {
        return title ?
                this.booksList.filter((book) => new RegExp(title, 'gi').test(book.title)) :    [];
    }

    getBook (isbn: number) {
        var selectedBook = this.booksList.filter(book => book.isbn === isbn);
        return selectedBook[0];
    }

    deleteBook (isbn: number) {
        this.booksList = this.booksList.filter(book => book.isbn !== isbn);
        return this.booksList;
    }

}
```

Infine, il tag di destinazione su `index.html`

```
    <books-list [books]="filteredBooks"></books-list>
```

Questo piu o meno è il risultato ☺



Fino a qui abbiamo implementato la classica ricerca su una base dati. Ovviamente potremmo prendere questo codice senza capire cosa è e cosa fa, sappiamo che per fare una ricerca si usa questo set di istruzioni.
Vale la pena però prenderci un po di tempo e trasformare questo form in qualcosa di piu articolato, soprattutto esplorare e combinare gli operatori per osservare i risultati. I campi di applicazione vedrai che risulteranno tantissimi e risolveranno un mare di casi che richiederebbero tonnellate di codice Javascript. Quindi prenditi tempo e non andare oltre fino a quando non ti funzionano implementazioni diverse da questo esempio

Capitolo 4 – JavaScript ES7, l'evoluzione

Ritengo che sia importante soffermarsi sull'evoluzione che ha fatto JavaScript nel tempo ed esporre alcuni concetti fondamentali che servono per apprendere bene Angular, perche una delle cose che ho avuto meno chiare all'inizio è stato distinguere bene che cosa è puro JavaScript, cosa è Angular, cosa è TypeScript. E' facile fare confusione.

Partiamo da un cenno alla storia del linguaggio:

1995: Nasce **JavaScript** con nome LiveScript. Con questo termine oggi non ci rivolgiamo a una specifica versione dello standard, ma piuttosto al suo utilizzo a livello globale o parziale.
1997: vengono stabiliti gli standard **ECMAScript**, un linguaggio standardizzato dal ECMA International e supervisionato dal comitato T39
1999: Esce ES3 e IE5 è di gran moda
2000–2005: **XMLHttpRequest**, a.k.a. **AJAX,** guadagna popolarità in app come OWA - Outlook Web Access (2000) e Oddpost (2002), Gmail (2004) e Google Maps (2005).
2009: Esce ES5 (che è quello che maggiormente usiamo oggi) con forEach, Object.keys, Object.create e gli standard JSON
2015: Esce **ES6/ECMAScript2015**; ha soprattutto rifiniture sintattiche perché non c'era pieno accordo nella community per far uscire qualcosa di più innovativo. ES6 è lo standard utilizzato da **NodeJS**
2016: Esce **ES7**, presenta un esiguo numero di funzionalità implementate e non dovrebbe comportare grossi problemi di compatibilità nei browser in grado di supportare la versione ES2015

ES5/6 - bignami

Costanti
parola chiave per definire le costanti: *const*
Con const l'elemento potrà essere soltanto letto ma non modificato.

```
const PI = 3.141593
```

let

Con la parola chiave let è possibile isolare la definizione di variabili al blocco corrente, evitando di sovraccaricare la memoria. Alla chiusura del blocco le variabili dichiarate con let saranno rimosse dal garbage collector.

```
for (let i = 0; i < 10; i++) {
  console.log(i) // 0 1 2 3 ... 9
}
```

closure

In ES5 per isolare lo scope di una funzione era necessario usare le closure (*function () {})(),* mentre con ES6 è possibile usare i blocchi delimitati dalle parentesi graffe {} come ambienti isolati.

```
{
  function num() {return 1}
  {
    function num() {return 5}
    console.log(num()) // 5
  }
  console.log(num()) // 1
```

```
}
```

Arrow functions

In Javascript le funzioni anonime sono uno dei costrutti più utilizzati ma ogni volta richiedono un minimo di boilerplate, ossia di scrittura di codice ripetitivo.

Con le arrow function di ES6, scrivere funzioni anonime diventa una passeggiata.

```
var numbers = [0,1,2,3,4]
var increments = numbers.map(num => ++num) // [1,2,3,4,5]
```

Parametri di default

Una grande mancanza delle versioni precedenti di JS riguarda i parametri delle funzioni.
Con ES6 è possibile inizializzarli con un valore di default, è possibile raggruppare una lista di parametri in uno solo ed è anche possibile trasformare un array in una lista di parametri.

```
// Default Params
function sum(x = 10, y = 20) {
  return x + y
}

console.log(sum()) // 30

// Rest Params
function sum(x, y, ...abc) { // ...abc = [30,40,50]
  let total = abc.reduce((a,b) => a + b) // 30+40+50 = 120
  return x + y + total // 10 + 20 + 120 = 150
}

console.log(sum(10, 20, 30, 40, 50)) // 150

// Spread Operator
let people = ['foo', 'bar', 'jar']
let others = ['pippo', 'pluto', ...people]
let numbers = [1,2,3,4,5]

console.log(sum(...numbers)) // 15
```

Template string

ES6 introduce le template string, ossia stringhe che permettono l'inclusione di variabili e valori, evitano così la tradizionale concatenazione di stringhe. *${<variabile>}*

```
let sentence = `
multiline string
with tabulation
and carriage returns
`

console.log(sentence)

let name = 'Mario'
let template = `hello ${name}`

console.log(template) // hello Mario
```

Metodi proprietà

I metodi proprietà degli oggetti assumono una nuova firma, riducendo l'ammontare di codice da scrivere.

```
let person = {
  introduce() {
    console.log('My name is Mario')
  },
  ['say' + 'My' + 'Name']() {
    console.log('Mario')
  }
}

person.introduce() // My name is Mario
person.sayMyName() // Mario
```

Destrutturazione

A volte può essere necessario ritornare più di un valore e di salvare tali valori in più variabili o in oggetti complessi: tramite la destrutturazione è possibile risparmiare codice e tempo nell'assegnazione di più risultati di ritorno dall'esecuzione di una funzione.

```
function people() {
  return ['foo', 'bar', 'jar']
}

let [personFoo, personBar, personJar] = people()
console.log(personFoo, personBar, personJar) // foo bar jar
```

Classi

Seguendo lo stile della programmazione orientata agli oggetti, ES6 introduce il nuovo sistema di classi. Questo permette di creare ed estendere classi, mantenendo però la struttura prototipata di ES5.

```
class Person {
  constructor(name, surname) {
    this.name = name
    this.surname = surname
  }

  introduce() {
    console.log(`Hi, my name is ${this.name} ${this.surname}`)
  }

  static type() {
    console.log("I'm a Person")
  }
}

class Developer extends Person {
  constructor(name, surname, skill) {
    super(name, surname) // Person.constructor
    this.skill = skill
  }

  introduce() { // override
    console.log(`Hi, my name is ${this.name} ${this.surname} and I know ${this.skill}`)
  }
}

let mario = new Person('Mario', 'Bros')
mario.introduce() // Hi, my name is Mario Bros
Person.type() // I'm a Person

let luigi = new Developer('Luigi', 'Bros', 'ES6') // inherits every props and methods, even those static
luigi.introduce() // Hi, my name is Luigi Bros and I know ES6
Developer.type() // I'm a Person
```

Moduli

Javascript ha sempre necessitato di un sistema a moduli, cosa già nota in diversi linguaggi.
Con ES6 viene introdotto il nuovo sistema per esportare ed importare moduli.

```
// file js/myModule.js
export function sum(x, y) {return x + y}
export const ADDENDUM = 10

// file yourModule.js
import * as myModule from "js/myModule"
import {sum, ADDENDUM} from "js/myModule"

console.log(myModule.sum(10, 20)) // 30
console.log(sum(ADDENDUM, ADDENDUM)) // 20
```

Capitolo 5

Forms – Creiamo insieme la WebApp FORM-DEMO

Form-demo app

Per esplorare quanto piu possibile le tecniche di gestione dei forms in Angular creeremo una nuova app su cui faremo tutte le prove e potremo vedere in azione il codice che descriviamo. Al di la dei form che è possibile gestire vedremo piu in dettaglio come fare il redirect, una gridview, inclusione di moduli custom e insomma, senza complicare troppo il programma cercheremo di infilarci quante piu nozioni possibili per una app reale.
Il consiglio è quello di approfittarne per creare le proprie varianti e vedere in azione il tuo codice. Il mio deve essere come riferimento.
Il codice su Plunker: https://plnkr.co/edit/SPQiohvgHQgGEhJdzWUt?p=preview
WebPackBin: https://www.webpackbin.com

In Angular 5 sono disponibili due tipi diversi di forms: **form-driven**,**form-reactive**,**nested-form**. Vedremo ogni tipo di template usando lo stesso esempio per vedere come le stesse cose possono essere implementate in modi diversi. Più avanti, con i **nested-form,** vedremo un nuovo approccio su come impostare e lavorare con forms nidificati.

In Angular 5 i seguenti quattro stati sono comunemente usati dai forms:

- `valid` - stato della validità di tutti i controlli sul form, true se tutti i controlli sono validi
- `invalid` - inverso di valid; true se alcuni controlli non sono validi
- `pristine` – (incontaminato) è uno stato sulla "pulizia" del form; è True se nessun controllo è stato modificato
- `sporco` - inverso di pristine; true se qualche controllo è stato modificato

Form-driven

Crea un nuovo progetto `ng new form-demo`

Inserisci bootstrap nel `package.json` sotto `dependancies`

```
    "bootstrap": "4.0.0-alpha.4",
    "font-awesome": "4.7.0",
    "web-animations-js": "^2.3.1",
```

Sostituisci il file `index.html`

```
<!DOCTYPE html>
<html>

  <head>
    <base href="." />
    <title>angular2 playground</title>
    <link href="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0-alpha.6/css/bootstrap.min.css" rel="stylesheet">
    <link rel="stylesheet" href="style.css" />
    <script src="https://unpkg.com/core-js@2.4.1/client/shim.min.js"></script>
    <script src="https://unpkg.com/zone.js/dist/zone.js"></script>
    <script src="https://unpkg.com/zone.js/dist/long-stack-trace-zone.js"></script>
    <script src="https://unpkg.com/reflect-metadata@0.1.3/Reflect.js"></script>
    <script src="https://unpkg.com/systemjs@0.19.31/dist/system.js"></script>
    <script src="config.js"></script>
    <script>
    System.import('app')
      .catch(console.error.bind(console));
  </script>
  </head>

  <body>
    <my-app>
    loading...
```

```
  </my-app>
  </body>

</html>
```

In \src\app

app.component.ts

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

@Component({
  selector: 'my-app',
  templateUrl: 'app.component.html'
})
export class AppComponent {

}
```

app.moudule

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

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

e il nostro form **app.component.html**

```
<form>
  <div>
    <label>Name</label>
    <input type="text" name="name"/>
  </div>

  <div>
    <label>Birth Year</label>
    <input type="text" name="birthYear"/>
  </div>

  <div>
    <h3>Location</h3>
    <div>
      <label>Country</label>
      <input type="text" name="country"/>
    </div>
    <div>
      <label>City</label>
      <input type="text" name="city"/>
    </div>
  </div>

  <div>
    <h3>Phone numbers</h3>
    <div>
      <label>Phone number 1</label>
      <input type="text" name="phoneNumber[1]"/>
      <button type="button">remove</button>
    </div>
    <button type="button">Add phone number</button>
  </div>

  <button type="submit">Register</button>
  <button type="button">Print to console</button>
</form>
```

`npm start` e vedi che il browser mostri il nostro form correttamente

Name

Birth Year

Location

Country

City

Phone numbers

Phone number 1 remove

Add phone number

Register Print to console

Arricchiamolo di funzionalià

Aggiungiamo la direttiva **ngForm** in aggiunta a un id del form #

```
<form #myForm="ngForm">
```

E poi aggiungiamo la direttiva **ngModel** ai campi di Input

```
..
<input type="text" name="name" ngModel/>
..
```

ora tutti gli input sono registrati nel componente **ngForm**. Con questo abbiamo definito un modulo Angular 5 completamente funzionante ma non abbiamo ancora modo di accedere al componente e alle funzionalità che offre. Le due funzionalità principali di `ngForm` sono:

- Accesso ai valori dei controlli registrati
- Accesso allo stato dei controlli registrati

Ora informiamo il form che vogliamo controllare il JSON generato dal form stesso

```
<form #myForm="ngForm">
  ..
  <pre>{{myForm.value | json}}</pre>
```

`{{myForm.value | json}}` contiene il JSON fisico nel formato appropriato

A questo punto potremmo avere la necessita di avere piu numeri di telefono o localita. Per implementare un elemento come gruppo di valori va utilizzata la direttiva `ngModelGroup`.

Modifichiamo il controllo all'altezza del phonenumber nel form

```
<div ngModelGroup="phoneNumbers">
    <h3>Phone numbers</h3>
    <div *ngFor="let phoneId of phoneNumberIds; let i=index;">
      <label>Phone number {{i + 1}}</label>
      <input type="text" name="phoneNumber[{{phoneId}}]" #phoneNumber="ngModel" ngModel/>
      <button type="button" (click)="remove(i); myForm.control.markAsTouched()">remove</button>
    </div>
    <button type="button" (click)="add(); myForm.control.markAsTouched()">Add phone number</button>
</div>
```

Abbiamo quindi dichiarato che phonenumbers è un `ngModelGroup` ed effettuato un ciclo di lettura valori da **phoneNumbersIds** con `*ngFor`.

Ripetiamo la stessa cosa per **Locations**

L'istruzione `myForm.control.markAsTouched()` informa Angular che lo stato del Form è cambiato. Nota bene, **del form**, non del controllo. Questo gli serve per capire se durante il processo di validazione è effettivamente cambiato qualcosa o se può saltarlo.

Submit

l'**ngForm** con id `myForm` può essere passato come argomento alla funzione che servirà come gestore per l'evento `onSubmit` del modulo. Per una migliore integrazione l'evento `onSubmit` viene incapsulato da un evento specifico Angular 4, l'**ngSubmit**. Questa direttiva ci permette di avere il controllo di quello che stiamo per inviare al server:

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

Dopodichè nell'AppComponent dobbiamo implementare la funzione register()

```
import {NgForm} from '@angular/forms'
..
export class AppComponent {
..
register (myForm: NgForm) {
    console.log('Successful registration');
    console.log(myForm);
  }
..
}
```

Prova il pulsante ‘register’ e verifica che sulla console del browser appaia

Successful registration app.component.ts:31:4
Object { submitted: true, _directives: Array[6], ngSubmit: Object, form: Object }

Validazione

La validazione è veramente importante, esigenza che non nasce certo con Angular, ma vediamo come viene gestita in questo contesto. Lo scopo è quello di voler convalidare sempre l'input dell'utente (perche non ci fidiamo) per impedire l'invio o il salvataggio di dati non validi e dobbiamo mostrare qualche messaggio significativo sull'errore per guidare correttamente l'utente a immettere dati validi.

Per alcune regole di convalida da applicare su un determinato input, il corretto validatore deve essere associato all’input specifico. Angular 4 offre una serie di validatori comuni come: `required`, `maxLength`, `minLength` ...

Quindi, come possiamo associare un validatore con un input? Beh, abbastanza facile; basta aggiungere la direttiva del validatore al controllo:

```
    <input type="text" name="name" ngModel required/>
```

Aggiungiamo `novalidate` al form, che informa Angular di non utilizzare la validazione nativa del browser

```
<form #myForm="ngForm" (ngSubmit)="register(myForm)"  novalidate>
<p>Form validated: {{myForm.valid}}</p>
```

E aggiungiamo anche il controllo di validità del form `{{myForm.valid}}</`

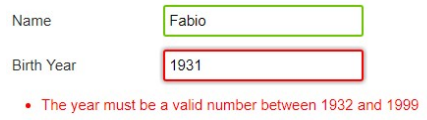
poi vogliamo che il campo `name` sia obbligatorio (va aggiunto required e l’id associato a `ngModel`)

```
  <input type="text" name="name" #name="ngModel" ngModel required/>
```

Inoltre vogliamo fare in modo che quando il campo di input è validato colora il bordo di verde, se ci sono errori di rosso. Dichiariamo due stili prima del tag <form ..>

```
<style>
  input.ng-dirty.ng-invalid { border: 2px solid #ff0000 }
  input.ng-dirty.ng-valid { border: 2px solid #6bc502 }
</style>
```

Dice ad Angular che cambi il colore di tutti i campi di input se contengono un valore. `ng-invalid` quando non è validato, `ng-valid` quando è valido



a questo punto vogliamo visualizzare un messaggio se ci sono errori. A questo scopo Implementiamo il componente `show-errors.component` nella cartella show-errors

```
// show-errors.component.ts
import { Component, Input } from '@angular/core';
import { AbstractControlDirective, AbstractControl } from '@angular/forms';

@Component({
 selector: 'show-errors',
 template: `
   <ul *ngIf="shouldShowErrors()">
     <li style="color: red" *ngFor="let error of listOfErrors()">{{error}}</li>
   </ul>
 `,
})
export class ShowErrorsComponent {

 private static readonly errorMessages = {
   'required': () => 'This field is required',
   'minlength': (params) => 'The min number of characters is ' + params.requiredLength,
   'maxlength': (params) => 'The max allowed number of characters is ' + params.requiredLength,
   'pattern': (params) => 'The required pattern is: ' + params.requiredPattern,
   'years': (params) => params.message,
   'countryCity': (params) => params.message,
   'uniqueName': (params) => params.message,
   'telephoneNumbers': (params) => params.message,
```

```
  'telephoneNumber': (params) => params.message
};

@Input()
private control: AbstractControlDirective | AbstractControl;

shouldShowErrors(): boolean {
  return this.control &&
    this.control.errors &&
    (this.control.dirty || this.control.touched);
}

listOfErrors(): string[] {
  return Object.keys(this.control.errors)
    .map(field => this.getMessage(field, this.control.errors[field]));
}

private getMessage(type: string, params: any) {
  return ShowErrorsComponent.errorMessages[type](params);
}

}
```

L'export del modulo è ShowErrorsComponent, quindi dobbiamo dichiararlo nell'app.module

```
..
import { ShowErrorsComponent } from './show-errors/show-errors.component';
..
  declarations: [ AppComponent,ShowErrorsComponent],
..
```

Per far visualizzare l'eventuale errore associato l'input name è sufficiente aggiungere il tag appropriato

```
    <input type="text" name="name" #name="ngModel" ngModel required/>
    <show-errors [control]="name"></show-errors>
```

Come si può vedere il nostro `show-error.component.ts` è valido per tutti i campi, basta cambiargli l'id sul `[control]="<id>"`

Validatori personalizzati

Uno dei casi piu tipici è quando le funzioni messe a disposizione del linguaggio non soddisfano le esigenze ed è necessario crearne di propri. Con Angular è possibile creare validatori custom, vediamone un esempio. Vogliamo estendere i controlli su BirthDate in modo che il range di valori sia da 1932 a 1999

Creiamo la cartella src\validators e creiamo il file vuoto `birth-year-validator.directive.ts`
In app.module dichiariamo l'oggetto **BirthYearValidatorDirective**

```
import { BirthYearValidatorDirective } from './validators/birth-year-validator.directive';
..

  declarations: [AppComponent, BirthYearValidatorDirective]
```

nel componente html aggiungiamo il validatore nel `ngForm`

```
<form #myForm="ngForm" (ngSubmit)="register(myForm)" years novalidate >
```

Infine come validatore inseriamo questo codice `birth-year-validator.directive.ts`

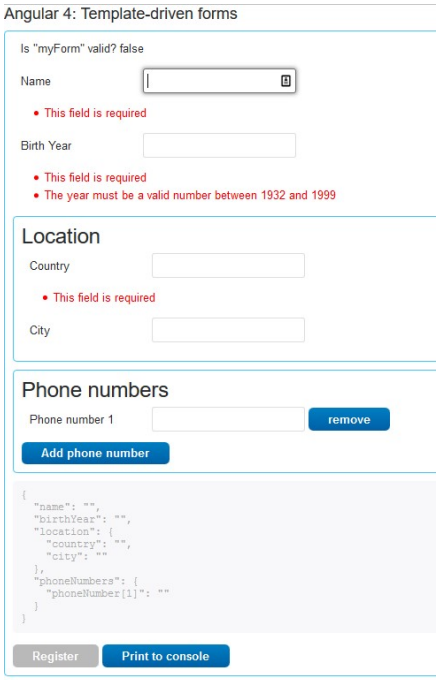
```
import { Directive } from '@angular/core';
import { NG_VALIDATORS, FormControl, Validator, ValidationErrors } from '@angular/forms';

@Directive({
  selector: '[birthYear]',
  providers: [{provide: NG_VALIDATORS, useExisting: BirthYearValidatorDirective, multi: true}]
})
export class BirthYearValidatorDirective implements Validator {

  validate(c: FormControl): ValidationErrors {
    const numValue = Number(c.value);
    const currentYear = new Date().getFullYear();
    const minYear = currentYear - 85;
    const maxYear = currentYear - 18;
    const isValid = !isNaN(numValue) && numValue >= minYear && numValue <= maxYear;
    const message = {
      'years': {
        'message': 'The year must be a valid number between ' + minYear + ' and ' + maxYear
      }
    };
    return isValid ? null : message;
  }
}
```

Ci sono alcune cose da spiegare qui. In primo luogo si può notare che abbiamo implementato la nostra interfaccia **Validator**. Il metodo di convalida controlla se l'utente è compreso tra i 18 ei 85 anni dall'anno di nascita inserito. Se l'input è valido, viene restituito **null** oppure un oggetto contenente il messaggio di

convalida. l'ultima e più importante è dichiarare questa direttiva come un **Validator**. Si fa attraverso il parametro " `providers` " del decoratore @Directive. Questo validatore è fornito come un valore del multi-provider **NG_VALIDATORS**. Inoltre, non dimenticare di dichiarare questa direttiva nel **ngModule**:

```
    <input type="text" name="birthYear" #year="ngModel" ngModel required birthYear>
    <show-errors [control]="year"></show-errors>
```

Abbiamo quindi introdotto alcune novità:
Il decoratore @Directive che specifica il tag di destinazione nel **selector** e i **providers** utilizzati. Nota bene i parametri **multi** e **useexisting** nei providers. La prima dice ad Angular che questa non è l'unico controllo di validazione da elaborare, useextisting specifica il nome della classe contenente il codice di validazione. Ricorda di utilizzare questa sintassi quando crei un tuo validatore. In realtà ti dovrebbe essere sufficiente copiare e adattare questi snippets



Reactive Forms

Forms reattivi o Form programmatici o Module-driven, sono le diverse nomenclature con cui ci si riferisce a questa tecnica che esiste da Angular4 in poi. Non c'è nella 2. Per fare delle prove puoi utilizzare la stessa app che abbiamo usato per i form-driven. Devi solo aggiungere un'altra pagina e gestirla con il routing, che è spiegato nel capitolo successivo

Partiamo con aggiornare l'app.module

```
import {ReactiveFormsModule} from '@angular/forms'
..
@NgModule({
  imports: [BrowserModule, FormsModule, ReactiveFormsModule, AppRoutingModule],
..
```

Poi il form

```
<style>
  input.ng-dirty.ng-invalid { border: 2px solid #ff0000 }
  input.ng-dirty.ng-valid { border: 2px solid #6bc502 }
</style>

<h1>Angular 4: REACTIVE Forms</h1>
<form [formGroup]="myForm" (ngSubmit)="register(myForm)" countryCity telephoneNumbers novalidate>
  <div>
    <label>Name</label>
    <input type="text" name="name" formControlName="name" required uniqueName>
    <show-errors [control]="myForm.controls.name"></show-errors>
```

```
  </div>

  <div>
    <label>Birth Year</label>
    <input type="text" name="birthYear" formControlName="birthYear" required birthYear>
    <show-errors [control]="myForm.controls.birthYear"></show-errors>
  </div>

  <div formGroupName="location">
    <h3>Location</h3>
    <div>
      <label>Country</label>
      <input type="text" name="country" formControlName="country" required>
      <show-errors [control]="myForm.controls.location.controls.country"></show-errors>
    </div>
    <div>
      <label>City</label>
      <input type="text" name="city" formControlName="city">
    </div>
  </div>

  <div formArrayName="phoneNumbers">
    <h3>Phone numbers</h3>
    <div *ngFor="let phoneNumberControl of myForm.controls.phoneNumbers.controls; let i=index;">
      <label>Phone number {{i + 1}}</label>
      <input type="text" name="phoneNumber[{{phoneId}}]" [formControlName]="i" required telephoneNumber>
      <show-errors [control]="phoneNumberControl"></show-errors>

      <button type="button" (click)="remove(i)">remove</button>
    </div>
    <button type="button" (click)="add()">Add phone number</button>
  </div>

  <pre>{{myForm.value | json}}</pre>
  <button type="submit">Register</button>
  <button type="button" (click)="printMyForm()">Print to console</button>
</form>
```

E il ts

```
import { FormGroup, FormControl, FormArray, NgForm  } from '@angular/forms';
import { Component, OnInit } from '@angular/core';

@Component({
  selector: 'dummy-tag',
  templateUrl: './form-reactive.component.html',
  styleUrls: ['./form-reactive.component.css']
})
export class FormReactiveComponent implements OnInit {

  private myForm: FormGroup;

   constructor() {
   }

   ngOnInit() {
     this.myForm = new FormGroup({
       'name': new FormControl(),
       'birthYear': new FormControl(),
       'location': new FormGroup({
         'country': new FormControl(),
         'city': new FormControl()
       }),
       'phoneNumbers': new FormArray([new FormControl('')])
     });
   }

   remove(i: number) {
    (<FormArray>this.myForm.get('phoneNumbers')).removeAt(i);
   }

   add() {
    (<FormArray>this.myForm.get('phoneNumbers')).push(new FormControl(''));
   }

   printMyForm() {
     console.log(this.myForm);
   }

   register(myForm: NgForm) {
     console.log('Registration successful.');
     console.log(myForm.value);
   }

 }
```

Tutto il resto, validatori, routings, showerrors ecc sono identici al form-driven.
Confrontando il TypeScript dei due tipi di forms si nota subito qual è la sostanziale differenza. Nei form reattivi l'oggetto Input (o qualsiasi altro tag html) è creato a run time ed è disponibile a livello di codice per farci tutto quello che in jQuery si faceva con l'addressing by id o class.
I controlli fanno parte di un FormGroup e sono dinamici, mentre nel form-driven sono statici.
A occhio raccomanderei di utilizzare se possibile sempre la seconda soluzione, mi sembra piu versatile

Nesting Forms

I Forms nidificati sono utili in alcuni quando è necessario determinare lo stato (ad esempio validità) di un sottogruppo di controlli. Pensate ad un albero di componenti; possiamo essere interessati alla validità di un determinato componente nel mezzo di quella gerarchia. Questo sarebbe davvero difficile da ottenere se avevamo una singola forma nella componente principale. Ma, oh ragazzo, è un modo sensibile su un paio di livelli. Innanzitutto, non è consentito l'inserimento di forme HTML effettive, secondo le specifiche HTML. Potremmo cercare di nidificare elementi <form>. In alcuni browser potrebbe effettivamente funzionare, ma non possiamo essere sicuri che funzionerà su tutti i browser, in quanto non è nella specifica HTML. In AngularJS, il modo per aggirare questa limitazione è stato quello di utilizzare la direttiva ngForm, che ha offerto le funzionalità di modulo AngularJS (solo raggruppamento dei controlli, non tutte le funzionalità di forma come la pubblicazione sul server), ma potrebbe essere posto su qualsiasi elemento. Inoltre, in AngularJS, l'annidamento di forme (quando dico forme, intendo NgForm) era disponibile fuori dalla scatola. Solo dichiarando un albero di due elementi con la direttiva ngForm, lo stato di ogni forma è stato propagato verso l'alto all'elemento radice.

Nella prossima sezione, avremo un'occhiata a una coppia opzioni su come nascondere le forme. Mi piace sottolineare che possiamo distinguere due tipi di nidificazione: all'interno dello stesso componente e tra i diversi componenti.

Modifichiamo l'html del form-driven (il primo form che abbiamo fatto)

```
<div ngModelGroup="location" #location="ngModelGroup" countryCity>
```

Abbiamo aggiunto al `div` della `location`, la keyword `#location="ngModelGroup"` e la direttiva del validatore `countryCity`

Lo stesso lo facciamo sul div del telefono

```
<div ngModelGroup="phoneNumbers" #phoneNumbers="ngModelGroup" telephoneNumbers>
```

Links

Working with Angular 4 Forms: Nesting and Input Validation (Articolo originale di Igor Geshoski)

Sorgenti

Angular 4: Template-driven forms (Originale)

Angular 4: Reactive forms

Nestable forms in Angular 4 su plnkr.co (Originale dell'autore)

Capitolo 5
Routing e Direttive di Attributo

Routing

L'idea di avere una master page e visualizzare diverse pagine oppure un conainer che visualizza diversi content selezionandole da un menu <nav> fisso è un aspetto tipico di qualsiasi webapp. Uno dei modi per farlo in Angular è attraverso il routing, o indirizzamento pagine. Esso consente di effettuare una **mappatura** tra URL, componenti che fanno parte di un'applicazione e view in maniera lineare ed efficace.
Fare il routing Non è una cosa difficile ma non ci si arriva per intuito perche ci sono delle caratteristiche fisse e una struttura molto rigida da rispettare. Conviene prendere la base direttamente da qui e adattarla ai tuoi contenuti. Segui passo passo

Supponendo di avere una home Index.html e due pagine Page1.html e Page2.html strutturalmente vengono innestate in questo modo
`Index.html` – ci va il tag **`<app-root><\app-root>`** che conterrà tutto
`app.component.html` – ci vanno i due pulsanti "apri page1" e "apri page2" e **`<router-outlet></router-outlet>`** che conterra o page1 o page2
`page1.html` – tag page1
`page2.html` – tags page2

1) assicurati che index.html contenga il tag **`<app-root><\app-root>`**

2) crea i componenti page1.html e page2.html, posizionati sulla root e esegui il comando

```
ng generate component page1
ng generate component page2
```

3) crea il file di routing, posizionati sulla root e esegui il comando

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

Crea **`app-routing.module.ts`** (Se non è uguale a questo copia e incolla questo)

```
import { NgModule }             from '@angular/core';
import { RouterModule, Routes } from '@angular/router';
import { Page1Component } from './page1/page1.component';
import { Page2Component } from './page2/page2.component';

const routes: Routes = [
  { path: 'page1', component: Page1Component },
  { path: 'page2', component: Page2Component },
  { path: '', redirectTo: '/page1', pathMatch: 'full' }
];

@NgModule({
  exports: [ RouterModule ],
  imports: [ RouterModule.forRoot(routes) ]
})
export class AppRoutingModule {}
```

la variabile routes contiene tutti i percorsi mappati, nell'import specifichiamo **forRoot()** al modulo RouterModule. Il metodo **forRoot()** permette di associare l'array di `routes` passato come argomento, alla root dell'applicazione. Esso esegue la navigazione iniziale partendo dall'URL corrente del browser.

4) modifica app.module

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

import { AppComponent } from './app.component';
import { AppRoutingModule } from './/app-routing.module';
import { Page1Component } from './page1/page1.component';
import { Page2Component } from './page2/page2.component';
```

```
@NgModule({
  declarations: [
    AppComponent,
    Page1Component,
    Page2Component
  ],
  imports: [
    BrowserModule,
    AppRoutingModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule {
}
```

5) Metti i links su app.component.html (dove vuoi)

```
<nav>
    <a routerLink="/page1">Apri Pagina 1</a>
    <a routerLink="/page2">Apri Pagina 2</a>
</nav>
<router-outlet></router-outlet>
```

Nota Bene: *router-outlet è un tag fisso di sistema, senza questo non funziona nulla. Ricorda che i selector nei componenti page1 e page2 vengono ignorati in quanto il routing li destina su questo tag automaticamente*

6) Verifica **`app.component.ts`** **(questo va su app.root correttamente)**

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

@Component({
  selector: 'app-root',
  templateUrl: 'app.component.html'
})
export class AppComponent {
  title = 'Test Forms in Angular 5';
}
```

Ecco fatto, senza toccare altro dovresti vedere visualizzata come default la page1, se clicchi sul *Apri Pagina 2* vedi la 2 sulla 1

Direttive di Attributo

Definiremo ora alcune direttive personalizzate che andranno arricchire quelle di Angular. Angular ha le direttive piu comuni, ma non possono da sole coprire tutti i casi specifici. In alcuni di questi siamo costretti a crearne delle nostre.
Abbiamo visto a fondo le *Direttive di validazione* nel capitolo precedente, ora vediamo le *Direttive di attributo*, che sono ancora piu semplici e permettono di cambiare comportamento o aspetto di un singolo elemento. Successivamente vedremo **`Direttive strutturali`**, che servono a modificare il layout dell'thml.
Per il momento creiamo una direttiva personalizzata che riproduca una gia esistente di Angular.
Ovviamente non ha utilità pratica ma serve a capire come funzionano

Cosa sono?	Una direttiva di attributo è un set di istruzioni TypeScript che cambiano stile o classe a un determinato tag sulla pagina, per un dato evento
Perché sono utili?	Una volta ben comprese danno la possibilità di manipolare gli stili a runtime con poche istruzioni fanno cose altrimenti complesse
Punti deboli o limiti?	L'unico problema è che si è portati a scrivere codice invece di utilizzare proprie

caratteristiche ‘di configurazione direttiva’ come Proprieta di Input e Output e @HostListener

Ci sono alternative? Angular supporta altri due tipi di Direttive, strutturali e componenti che a volte possono essere più adatti a un dato compito. Puoi non usarle ma il rischio è che ti trovi HTML o javascript complesso difficile da leggere e da mantenere. Come quantità di codice le Direttive di Attributo non hanno paragoni in quanto a compattezza

Aggiungiamo una pagina a form-demo

Utilizziamo l’app **form-demo** che abbiamo implementato in precedenza. Poiché abbiamo gia il routing configurato possiamo facilmente aggiungere pagine per i nostri esempi futuri.
Associamo un colore al mouseover diverso a seconda del colore scelto. In pratica vogliamo che un elemento *<p> passa sopra con il mouse <\p>* assuma il colore diverso al passaggio del mouse. Ora vediamo come fare, basti tenere presente che allo stesso modo è possibile cambiare qualsiasi altro attributo di stile per qualsiasi altro evento o condizione.
Ovviamente se volessimo un semplice cambio di colore al mouseover utilizzeremo una classe di stile con `:hover` e sarebbe semplice. Qui dobbiamo prendere il colore da uno di tre radio button selezionati. A occhio un po di annidamenti jQuery sarebbe inevitabile.

Aggiungi una cartella `\pages`, in questa ci metteremo altri esempi. Posizionati con il prompt dos

```
ng generate component form-directives
ng generate directive highlight
```

```
PS C:\0\websites\Angular\form-demo\src\app\pages > cd..
PS C:\0\websites\Angular\form-demo\src\app\pages> ng generate component form-directives
  create src/app/pages/form-directives/form-directives.component.html (34 bytes)
  create src/app/pages/form-directives/form-directives.component.spec.ts (685 bytes)
  create src/app/pages/form-directives/form-directives.component.ts (345 bytes)
  create src/app/pages/form-directives/form-directives.component.css (0 bytes)
  update src/app/app.module.ts (1716 bytes)
PS C:\0\websites\Angular\form-demo\src\app\pages> cd .\form-directives\
PS C:\0\websites\Angular\form-demo\src\app\pages\form-directives> ng generate directive highlight
  create src/app/pages/form-directives/highlight.directive.spec.ts (236 bytes)
  create src/app/pages/form-directives/highlight.directive.ts (147 bytes)
  update src/app/app.module.ts (1822 bytes)
PS C:\0\websites\Angular\form-demo\src\app\pages\form-directives>
```

form-directive.component.html

```
<h1>My First Attribute Directive</h1>

<h4>Pick a highlight color</h4>
<div>
  <input type="radio" name="colors" (click)="color='lightgreen'">Green
  <input type="radio" name="colors" (click)="color='yellow'">Yellow
  <input type="radio" name="colors" (click)="color='cyan'">Cyan
</div>
<p [appHighlight]="color">Highlight me!</p>

<p [appHighlight]="color" defaultColor="violet">
  Highlight me too!
</p>

<hr>
<p><i>Mouse over the following lines to see fixed highlights</i></p>

<p [appHighlight]="'yellow'">Highlighted in yellow</p>
<p appHighlight="orange">Highlighted in orange</p>
```

Questa pagina la creiamo per l'esempio ma in una app reale abbiamo gia un controllo sulla pagina su cui vogliamo creare la direttiva. Qundi html e ts del componente ce lo abbiamo gia

form-directive.component.ts

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

@Component({
  selector: 'dummy-tag',
  templateUrl: './form-directives.component.html'
})
export class FormDirectivesComponent {
  color: string;
}
```

Nel ts del componente ci serve solo aggiugere una variabile color di tipo string.

E ora scriviamo la direttiva vera e propria
highlight.directive.ts

```ts
import { Directive, ElementRef, HostListener, Input } from '@angular/core';

@Directive({
  selector: '[appHighlight]'
})
export class HighlightDirective {

  constructor(private el: ElementRef) { }

  @Input() defaultColor: string;

  @Input('appHighlight') highlightColor: string;

  @HostListener('mouseenter') onMouseEnter() {
    this.highlight(this.highlightColor || this.defaultColor || 'red');
  }

  @HostListener('mouseleave') onMouseLeave() {
    this.highlight(null);
  }

  private highlight(color: string) {
    this.el.nativeElement.style.backgroundColor = color;
  }
}
```

Perche sia visibile nella app la direttiva va inclusa nell'app.module.ts

```ts
import { HighlightDirective } from './pages/form-directives/highlight.directive';
..
declarations:
  [ ..,
    HighlightDirective
  ],
```

Vediamo gli import di questa direttiva

- **Directive** fornisce la funzionalità del decoratore **`@Directive`**.
- **ElementRef** fornisce l'accesso diretto all'elemento sul DOM.
- **Input** consente di vedere l'elemento all'interno dell'html come stringa

poi la **`@Directive`** contiene il nome di fantasia che diamo al selettore. Sull'html ci riferiremo a questo nome. Insomma è il nome della direttiva sull'html: `[AppHighlight]`.

La classe della direttiva invece si chiama **HighlightDirective**
Il costruttore passa l'oggetto del DOM che contiene [AppHighlight], ovvero l'intero <p><\>. Se il tag è ripetuto l'elemento è una array di oggetti.

Poi il primo input prende l'eventuale attributo defaultColor presente nel <p defaultcolor="black" ><\p>

```ts
@Input() defaultColor: string;
```

il secondo @Input prende l'attributo direttiva e lo mette in una variabile `HighLightColor`

```ts
@Input('appHighlight') highlightColor: string;
```

Come si può vedere nell' html di esempio, può contenere "color", in questo caso prenderà il colore contenuto della variabile color, altrimenti se è un colore lo passa alla direttiva e lo renderizza sempre.

Poi dobbiamo descrivere quello che succede quando il mouse entra e quando esce. @HostListener fa al caso nostro, è il decoratore che intercetta l'evento descritto fra parentesi e lo passa alla funzione, che qui abbiamo chiamato MouseEnter()

```
@HostListener('mouseenter') onMouseEnter() {
  this.highlight(this.highlightColor || this.defaultColor || 'red');
}
```

L'istruzione che esegue è:
Se il primo valora è valorizzato prendi quello
Se no prendi il secondo
Se non c'è nulla assegna sempre rosso

Poi al mouseLeave dobbiamo annullare la direttiva e quindi togliamo il colore al controllo

```
@HostListener('mouseleave') onMouseLeave() {
  this.highlight(null);
}
```

Infine dichiariamo la funzione JavaScript che assegna il colore

```
private highlight(color: string) {
  this.el.nativeElement.style.backgroundColor = color;
}
```

this.el è l'oggetto o il gruppo di oggetti
in jQuery se non sbaglio sarebbe stato `$(".highlight").css("background-color",color);`

e alla fine visto che abbiamo capito aggiungiamone un altra

```
@HostListener('click') oClick() {
    this.el.nativeElement.classList.add("bg-success")
}
```

@ViewChild

Questo decoratore dice ad Angular di utilizzare le direttive attributo custom all'interno di ts.
Come abbiamo visto, la nostra direttiva personalizzata `highlight.directive` è visibile all'interno dell'html, il binding è nell'html. Per poterla usare nel componente ts va dichiarata con la `@ViewChild`

vedi il sorgente originale dalla guida di Angular adattato per il form-demo:
https://angular.io/generated/live-examples/attribute-directives/eplnkr.html

Aggiungere nostre librerie di Helper o Utilities

Nella struttura di un progetto non può mancare l'helper dove ci infiliamo tutte le funzioni di utility, quelle fighe usate per i progetti passati. Spesso ci portiamo dietro spazzatura che non useremo mai, ma dell'helper in un progetto non se ne puo fare a meno.

Quello che c'è da fare in Angular è importarlo nel collettore principale `app.module.ts` e renderlo accessibile al resto dei componenti.

Si fa in un modo solo, alle solite ☺ Faccio un esempio

\app\Utils.ts

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

@Injectable()
export class Utils {

     redirect(url:string) { window.location.href = url; }

     redirectFix() {
        window.location.href = 'http://www.google.com';
    }
}
```

Va dichiarato @Injectable

In app.module.ts va dichiarato come provider di 'qualcosa'

```
import { Utils } from './utils';
..
@NgModule({
 ..
  providers:  [ Utils ]
 ..
})
```

E quando la voglio usare in un componente qualsiasi..

nell'html (il button è un esempio)

```
<button (click)="redir()" alt="Apre url esterno usando un helper">Apri Google</button>
Oppure, ottimizzato
<button (click)="this.uti.redirectFix()" alt="Apre url esterno usando un helper">Apri Google</button>

<button (click)="this.uti.redirectFix()" alt="Apre url esterno usando un helper">Apri Google</button>
```

nel .ts

```
import { Utils } from './utils';
..
export class AppComponent {

  constructor(private uti: Utils){
  }

  redir() {
     this.uti.redirectFix();
  }
..
}
```

E' importante definirlo privato ne costruttore altrimenti l'html non lo vede

Non si puo fare, ad esempio

```
export class AppComponent {
  uti: Utils;
  redir() {
     this.uti.redirectFix();  <-- qui dice che this.uti non è definito
  }
```

Debugging Angular

Aggiungiamo un logger al nostro helper per il debugging

Se vogliamo dare un tocco di professionalità alla nostra app possiamo aggiungere un Logger che sia parametrico, che vada su console con warning, error o info.

~~logging su filesystem~~

..

Qui invece possiamo scrivere su console del browser con circospezione. Vediamo come:
Estendiamo la Utils

```
import { Component,Injectable } from '@angular/core';
import { environment } from '../environments/environment';

export let isDebugMode = environment.isDebugMode;
const noop = (): any => undefined;

@Injectable()
export class Utils {

     redirect(url:string) { location.href = url; }

     redirectFix() {

        console.log("entra") ;
        window.location.href = 'http://www.google.com';
    }
}
export abstract class Logger {
      info: any;
      warn: any;
      error: any;
}

@Injectable()
export class ConsoleLoggerService implements Logger {

      get info() {
        if (isDebugMode) {
          return console.info.bind(console);
        } else {
          return noop;
        }
      }

      get warn() {
        if (isDebugMode) {
          return console.warn.bind(console);
        } else {
          return noop;
        }
      }

      get error() {
        if (isDebugMode) {
          return console.error.bind(console);
        } else {
          return noop;
        }
      }

      invokeConsoleMethod(type: string, args?: any): void {
        const logFn: Function = (console)[type] || console.log || noop;
        logFn.apply(console, [args]);
      }
}
```

E nell'app.module lo dobbiamo dichiarare come nuovo Provider

```
import { Utils,ConsoleLoggerService } from './utils';
..
  providers:  [ Utils,ConsoleLoggerService ]
```

Con questo sistema possiamo debuggare qualsiasi componente, basta dichiararlo (nei component.ts, directive.ts. ecc)

```
import { Utils, ConsoleLoggerService} from '../../utils';
```

nel costruttore lo dichiariamo

```
constructor(private el: ElementRef,private logger: ConsoleLoggerService) { }
```

e nel listener della direttiva aggiungiamo il log del colore

```
@HostListener('click') oClick() {
    this.logger.info("Proprietà Color attiva: " + this.getColor());
    this.el.nativeElement.classList.add("bg-success")
  }
```

Questo è un esempio e lo possiamo importare in ogni ts, è visibile dall'html dentro {{ }}, insomma al posto del console.log("") abbiamo il nostro logger su cui possiamo scrivere il timestamp, calcolare tempi, ecc.

Debugging da Chrome

Usa `ng.probe()` nella console di chrome per eseguire effective debugging o usa Augury chrome extension che wrappa per te ng.probe()

Variabili di configurazione Environment

Una cosa che abbiamo introdotto implementando il logging è l'accesso a variabili in evnironment
Abbiamo aggiunto una proprietà **isDebugMode** al file `\evironments\environment`

```
export const environment = {
  production: false,
  isDebugMode: true
};
```

Lo abbiamo importanto e dichiarato nella Utils

```
import { environment } from '../environments/environment';

export let isDebugMode = environment.isDebugMode;
```

e valutato

```
        if (isDebugMode) {
          return console.info.bind(console);
        } else {
          return noop;
        }
```

Da qui è facile intuire come possiamo dotare la nostra webapp di parametri di configurazione.
Le piu tipiche sono gli endpoint api, gli host, username e password, secretkey, ecc ecc

Mappatura di Environment

Il CLI di Angular crea come predefiniti una cartella environment e all'interno due files evnironment.ts ed environment.prod.ts. E' facile intuire che uno è predisposto per le configurazioni in DEV e uno in Produzione. Questo file viene bindato al momento del build e chirarmante nel build ci finisce solo una versione. Per comodità lasciamo queste definizioni, ma se ad esempio volessimo aggiungere un altro livello di build tipo "staging" creeremo un altro file evnironment.staging.ts e alla fine lo mappiamo perche Angular sappia quale buildare. Sotto la keyword "app" avremo una cosa come questa.

```
..
    "environmentSource": "environments/environment.ts",
    "environments": {
      "dev": "environments/environment.ts",
      "staging": "environments/environment.staging.ts",
      "prod": "environments/environment.prod.ts"
    }
  }
..
```

Capitolo X

Risoluzione di problemi

In questo paragrafo vediamo come risolvere se si sputtana l'engine e va reinstallato da zero

- @angular/cli@1.4.9 (vedi nel package.json cosa hai) ha un bug nel link delle resource dalle pagine html e **non carica gli script o css dentro l'header html**. Va installata almeno la versione @1.5.0

Reinstallazione dei pacchetti

Disinstalla tutto quello che hai

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

cancella la cache manualmente e assicurati che sia tutto rimosso
su <tuo path user>\roaming\

```
C:\Users\pc1\AppData\Roaming\npm
```

Cancella
`\node_modules\@angular`
E tutte le cartelle `\node_modules\angular`
Cancella la cache
`\node_modules\node_cache\@angular*`
`\node_modules\node_cache\angular*`

E poi reinstalla @Angular

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

Dovresti avere le versioni sottostanti o superiori

Per controllare le versioni installate di nodeJS e dipendenze

```
npm list -global --depth 0
```

```
+-- @angular/cli@1.5.0
+-- cordova@7.0.1
+-- dotenv@4.0.0
+-- grunt-cli@1.2.0
+-- ionic@3.9.2
+-- nsp@2.7.0
+-- nvm@0.0.4
+-- nvmw@0.2.1
+-- oauth2-server@3.0.0
`-- tedious@2.0.0
```

Per verificare la versione di Angular-CLI – **Angular 5.0.1**

```
ng -v
```

```
To disable this warning use "ng set --global warnings.versionMismatch=false".
```

```
            |___/
Angular CLI: 1.5.0
Node: 8.4.0
OS: win32 x64
Angular: 5.0.1
... animations, common, compiler, compiler-cli, core, forms
... http, language-service, platform-browser
... platform-browser-dynamic, router

@angular/cli: 1.5.0
@angular-devkit/build-optimizer: 0.0.32
@angular-devkit/core: 0.0.20
@angular-devkit/schematics: 0.0.35
@ngtools/json-schema: 1.1.0
@ngtools/webpack: 1.8.0
@schematics/angular: 0.1.2
typescript: 2.4.2
webpack: 3.8.1
```

per verificare la versione di TypeScript

```
tsc -v
```

se ancora hai problemi a caricare script e css, almeno per quelli di sistema puoi prenderli online. Il problema è che puo essere piuttosto lento.

```
  <script src="https://unpkg.com/core-js@2.4.1/client/shim.min.js"></script>
  <script src="https://unpkg.com/zone.js/dist/zone.js"></script>
  <script src="https://unpkg.com/zone.js/dist/long-stack-trace-zone.js"></script>
  <script src="https://unpkg.com/reflect-metadata@0.1.3/Reflect.js"></script>
  <script src="https://unpkg.com/systemjs@0.19.31/dist/system.js"></script>
```

Se ti servono i tuoi custom e neanche con tutte queste madonne funziona, l'unica possibilità che ti resta, in attesa che il team angular sistemi il bug, è di bypassare il CLI, quindi devi dire a NodeJS di utilizzare gli script web-pack al posto del tool Angular-CLI.

Per farlo devi derivare il webpack partendo dal tuo –CLI. esegui

```
ng eject
```

Questo comando crea tutti i riferimenti webpack. Segui le istruzioni a video perche dovrai reinstallare alcuni packages che ti mancheranno. Al termine quindi esegui l'npm install

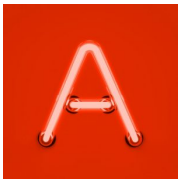
`Attenzione, prima di utilizzare ng eject` ti suggerisco di farti un backup della root e in piu tieni presente che l'avvio dell'app attraverso web-pack (dopo eject) impiega almeno 4 volte di piu del CLI, quindi eseguilo se non hai altre alternative

Per fare il roll-back e tornare al cli eliminando l'eject è semplice

Sul **`.angular-cli.json`** metti a false il parametro ejected:

```
..
"project": {
    "name": "master-book2-service",
    "ejected": false
  },
..
```

Novità di Angular 5



Logo ufficiale di Angular 5

Supporto PWA

Fino alla 4 lo sviluppo di applicazioni web progressive (PWA) era un processo elaborato e complesso. Bisognava prendersi cura sia nello sviluppo che nella distribuzione
Questa cosa cambia con Angular 5. Lo sviluppo di applicazioni web progressive è stato semplificato in modo tale da poter essere creato anche come impostazione predefinita. Ne beneficiano sia gli utenti che gli sviluppatori

Con Angular-CLI, Angular ha la capacità di creare configurazione e codice da solo. In sostanza, questo consente la creazione di applicazioni web mobili che dispongono di funzionalità di applicazioni mobili nativi, come le funzionalità offline, le notifiche push e un logo dell'applicazione nel menu di avvio di una particolare piattaforma. Internamente i service-worker vengono utilizzati per implementare la capacità offline e il trattamento delle notifiche push.

Angular fornisce i service-worker tramite il modulo @angular/service-worker. Altri elementi necessari sono la generazione del manifesto di cache HTML5 e la creazione di una shell di applicazioni. Considerando lo stato di connessione (offline / online) potrebbe essere necessario reagire agli eventi di routing all'interno dell'applicazione.

Il supporto PWA viene attivato dal seguente comando:

```
ng set apps.0.serviceWorker=true
```

Novità sui Forms Reattivi

La proprieta `updateOn` è passata al FormControl quando viene istanziato

```
this.email = new FormControl(null, { updateOn: 'blur' });
```

Se sono specificati validatori, lo sono anche come proprieta dell'oggetto:

```
this.email = new FormControl(null, {
    validators: Validators.required,
    updateOn: 'blur'
  });
```

Poi invece di specificare il comportamento di tutti i sotto-elementi del rispettivo FormControl viene fatto tramite FormGroup e FormArray. Dai un'occhiata a questo esempio di FormGroup che specifica la convalida per tutti gli elementi contenuti al momento della presentazione :

```
this.login = new FormGroup({
    email: new FormControl(),
    password: new FormControl()
  },
```

Gli elementi contenuti fanno l'override al comportamento

```
this.login = new FormGroup({
  email: new FormControl(null, {
     validators: Validators.required,
     updateOn: 'blur'
```

```
  }),
  password: new FormControl(null, [Validators.required])
}, {updateOn: 'submit'})
```

Utilizzo nei Form-Driven

Anche per i forms che sono definiti via template può essere passato il parametro updateOn. Questo si fa nel **`ngModelOptions:`**

```
<input type="email" ngModel [ngModelOptions]="{updateOn: 'submit'}">
```

Anche qui il setting è ereditato dal padre sul DOM

```
<form [ngFormOptions]="{updateOn: 'submit'}">
  <input name="email" ngModel type="email">
  <input name="password" ngModel type="email">
 </form>
```

Un elemento subordinato può sovrascrivere il valore di default cosi come fanno i forms reattivi

```
<form [ngFormOptions]="{updateOn: 'submit'}">
  <input name="email" ngModel type="email" [ngModelOptions]="{updateOn: 'blur'}">
  <input name="password" ngModel type="email">
 </form>
```

HttpClient

L'HttpClient introdotto con Angular 4.3 adesso supporta parametri per l'headers e per i params.

```
http.get("/api", {
  headers: {
      "X-DEMO-HEADER": "demo"
  },
  params: {
      "foo":  "bar"
  },
})
```

i18N Pipes

I pipes i18n che comprendono cose come data, numero, valuta e il simbolo percento sono stati influenzati da diverse modifiche interne. Ad esempio, l'Intl-API precedentemente utilizzato da Angular non è più supportato a causa delle incoerenze del browser. Invece, la localizzazione si basa sull'esportazione di dati dal repository di dati locale comune Unicode (CLDR).

Questi cambiamenti rappresentano una rottura con l'API. L'impostazione predefinita è US-English locale "en-US". Se hai bisogno di un'altra posizione o lingua, devi prima importarla. Ad esempio, è necessario importare il seguente codice per importare la lingua italiana "it" (equivalente a "it-IT" nell'International API) nell'appmodule:

```
import { registerLocaleData } from '@angular/common';
import localeIT from '@angular/common/locales/it';

registerLocaleData(localeIT);
```

quindi per le nostre app italiane a scanso di equivoci e sorprese sarà il caso di farlo..

Router

Il Router Angular è stato esteso con eventi aggiuntivi. Adesso per esempio, puoi creare un progress bar che si incrementa quando cambia il routing. Gli eventi in questione sono **`ActivationStart`** e **`ActivationEnd`** oppure **`ChildActivationStart`** e **`ChildActivationEnd`**. Ad esempio per creare uno spinner

```
router.events
// Event = RouteEvent | RouterEvent
.filter(e => e instanceof RouteEvent)
.subscribe(e => {
    if (e instanceof ActivationStart) {
        spinner.start();
    } else if (e instanceof ActivationEnd) {
```

```
        spinner.end()
    }
});
```

Animazioni

Le animazioni di sistema di Angular sono state incrementate con una serie di nuove caratteristiche :increment e :decrement per animare una transizione

```
@Component({
  animations: [
      trigger("counter", [
          transition(':increment', [ /*...*/ ]),
          transition(':decrement', [ /*...*/ ]),
      ])
  ],
  template: `
  <span [@counter]="count">
  `
})
class MyComponent() {
  count = 1;
}
```

In precedenza, quando una animazione era subordinata ad elementi era possibile solamente limitare la selezione a un certo numero di elementi che erano stati contati dall'inizio. Ora sono supportati anche limiti negativi, che si riferiscono all'ultimo elemento X. Qui un esempio che mostra come limitare gli ultimi 5 elementi con classe "person"

```
trigger("anim", [
  transition(":enter", [
      query(
          ".person",
          [ /** ... **/ ],
          { limit: -5 }
      )
  ])
])
```

Altre modifiche in Angular 5

Le API seguenti che erano state deprecate in Angular 4 sono state rimosse nella 5

- `OpaqueToken` è stata rimossa, al suo posto devi usare `InjectionToen`
- Il parametro del costruttore di `ErrorHandler` è stato rimosso
- `ngOutletContext` è stato rimosso da `ngTemplateOutlet`
- il parametro `initialNavigation` nel Router Angular prende solo i valori `"enabled"` e `"disabled"`. True, False, Legacy_enabled, legacy_disabled non sono piu ammessi
- l' `ng-container` deve essere usato al posto dei commenti i18n
- Il compilatore non usa più il `enableLegacyTemplate` come predefinito, va usato <ng-template>
- Al posto del modulo `angular/http` va usato `@angular/common/http`.
- `ReflectiveInjector` è deprecato, va usato StaticInjector e va usato `Injector.create` al posto di `ReflectiveInjector.resolveAndCreate`

Indice