

ANGULAR

100% Operativo

Da zero alla realizzazione di una Web APP, in 24 ore

Davide Copelli

Collana: CreareApp Vol.1

Copyright © DCOP
Via Santa Giustina 17 – 35030 Selvazzano Dentro (Padova)
Url: <https://www.dcopelli.it/> – Email: ing@dcopelli.it

Tutti i diritti riservati. Nessuna parte di questo libro può essere riprodotta con sistemi elettronici, meccanici o altri, senza l'autorizzazione dell'Editore.

Nome e marchi citati nel testo sono registrati alle rispettive case produttrici.

Editor: Indipendently Published
Redazione: Francesca e Chiara
Produzione: Davide C., Luca F., Massimo P.

Prima Edizione: luglio 2017
Seconda Edizione: febbraio 2018
Terza Edizione: marzo 2018
Quarta Edizione: settembre 2018
Quinta Edizione: giugno 2019
Sesta Edizione: maggio 2020
Settima Edizione: maggio 2021
Ottava Edizione: gennaio 2023

ISBN: 9781794295926

*A Cip per la pazienza,
a Mery per i succulenti pranzi,
a Giancarlo per la costanza!*

La semplicità è l'estrema perfezione
(Leonardo da Vinci)

Prefazione

Ciao e benvenuto al primo Corso Operativo su Angular, che ti aiuterà a capire come sfruttare questo framework per iniziare a sviluppare le tue prime semplici WebApp.

Mi chiamo Davide Copelli e sarò il tuo docente per le prossime 24 ore, o se preferisci, per le prossime 300 pagine circa!

Per chi fosse abituato a sentire la mia voce, non preoccuparti. A breve saranno online anche dei video riassuntivi per ogni capitolo, che potrai consultare nei portali indicati qui sotto, oltre che rimanere aggiornato sulle novità di Angular.

Per ulteriori approfondimenti, consulta i due siti qui sotto, di cui sono **fondatore**:

- [Video-Corsi.com](#): Il primo portale di e-learning nato in Italia, con MasterClass Operative sulle tecnologie web in Video
- [CreareApp.com](#): Il sito dedicato allo sviluppo di APP e non solo, con corsi, tutorial, laboratorio codice, riservata solo ai miei clienti!

Avvertenza importante da leggere!

Il libro, essendo "operativo", presuppone che tu abbia delle discrete basi di programmazione JavaScript, TypeScript, RxJS soprattutto per ciò che concerne la terminologia legata ai concetti di Oggetto, Classe, Observable, oltre che buone basi di HTML, CSS e linguaggi lato server. Nel libro faremo riferimento ad alcuni esempi con PHP.

Se ritieni di avere già queste conoscenze, allora puoi proseguire, diversamente ti consiglio di seguire qualche corso, o direttamente nel mio portale Video-Corsi.com o su CreareApp.com, dove potrai trovare articoli specifici su Angular e TypeScript.

Il libro è una sorta di guida operativa per **"rompere il ghiaccio" con Angular**, ed è stato volutamente concepito così. Non vuole essere un manuale su Angular e TypeScript e sulla relativa storia (ce ne sono fino troppi!), né è tantomeno un libro sulla programmazione, con tutta la teoria annessa (e se hai mai studiato qualche libro universitario, sai a cosa mi riferisco).

In sostanza, non troverai una sola riga che parli della storia di Angular, di cos'è un array, un framework MVC, una SPA, e tante parole che avrebbero farcito il libro di numerose pagine superflue per l'obiettivo che mi sono prefissato. Con semplici ricerche su Google, puoi colmare queste elementari lacune.

Questo approccio è quello ho voluto adottare nel 2004, quando ho fondato il portale [Video-Corsi.com](#), dopo alcuni anni aver conseguito una laurea magistrale in Ingegneria Elettronica a Padova, e dopo aver "consumato" decine di quaderni con appunti teorici.

Dalle decine di migliaia di commenti raccolti, posso affermare con una buona dose di sicurezza, che questa è la strada che preferiscono i miei discenti e spero anche tu.

Feedback o commenti

Per scrivere questo libro, mi sono "sporcato le mani" fin dalla prima versione di AngularJS (2010) e poi Angular (2015), con numerosi test e pratiche sul campo. La semplice applicazione che realizzeremo è stata sviluppata durante tre incontri in aula con sette professionisti del settore, tenuti a Padova, della durata di 8 ore circa, ciascuno: questo il motivo per cui nel titolo, ho inserito "da zero a 24 ore" (più o meno!).

Come dicevo, ho cercato di renderlo il più pratico e operativo possibile, tralasciando decine di pagine di approfondimenti tecnici, che per mia esperienza, spesso creano un *blocco mentale* nell'apprendimento di una nuova tecnologia ed è utile approfondire in un secondo momento. Nonostante questo, il libro supera le 300 pagine, questo per farti capire che "nulla è stato lasciato al caso"!

Sulla base di queste considerazioni, t'invito a lasciarmi un feedback su Amazon. Un commento ♥ POSITIVO a 4 o 5 stelle, non mi permetterà certo di diventare "milionario", ma è solo importante per il morale, quindi un grazie fin da ora.

Inoltre, so di non essere uno scrittore stile "Manzoni", quindi sentiti libero di scrivermi, preferibilmente in privato, per eventuali suggerimenti su come migliorare alcune frasi.

Prometto che citerò il tuo nome tra gli aiutanti revisori dello scritto sia nelle nuove versioni che usciranno sia in fondo alla descrizione del libro che trovi su Amazon (previo tuo consenso).

Ecco la mia email personale: ing@dcopelli.it e ricorda il mio mantra preferito, che rispolvero quando parlo o scrivo in pubblico:

"Prima di parlare domandati se ciò che dirai corrisponde a verità, se non provoca male a qualcuno, se è utile, ed infine se vale la pena turbare il silenzio per ciò che vuoi dire." (Gautama Buddha)

Codice degli esempi ed errata corrigere

Il codice lo puoi scaricare gratuitamente, seguendo le istruzioni che trovi in fondo al libro, nella sezione "Conclusioni, Codice lezioni e Video". Angular si aggiorna rapidamente, quindi è probabile che dall'acquisto del libro ai primi test, sia cambiato già qualcosa. Fai sempre riferimento al codice che trovi nella tua area privata (vedi fine libro).

Buona lettura e buon lavoro!

Davide Copelli {ing}

Indice

Che cosa realizzeremo	1
Capitolo 1 - Settings dell'ambiente di lavoro	
1.1 Da dove partire: l'ambiente di test locale	5
1.2 I migliori IDE per scrivere applicazioni	11
1.3 Comandi Utili CLI	14
Capitolo 2 - Il motore di un'app Angular	
2.1 La struttura di un'app	19
2.2 Il file index.html	22
2.3 Convenzioni per la progettazione del file index.html	24
2.4 Il mio primo Modulo: AppModule	25
Capitolo 3 - I componenti in Angular	
3.1 Il "cuore" di un'applicazione Angular	29
3.2 Il componente <app-root> creato in automatico	31
3.3 Come creare il componente Menu dell'app MetroChat	32
3.4 Tecniche alternative per creare un componente	34
3.5 Definire il nome del componente grazie al "selettore"	34
3.6 Analisi del componente <app-root>	36
3.7 Definire la rappresentazione grafica: Template	37
3.8 Come aggiungere un foglio di stile	39
3.9 Il Template del componente <app-root>	40
3.10 Definire la logica del componente	43
3.11 Popolare il Template con proprietà della classe	44
3.12 Definire i "segnaposto" nel Template con l'interpolazione	46
3.13 Inserire espressioni all'interno del "segnaposto"	47
3.14 Ciclo di vita di un'app Angular	48
Capitolo 4 - Manipolare il DOM con le direttive	
4.1 Ripetere elementi del DOM: la direttiva *ngFor	51
4.2 Sintassi della direttiva *ngFor	53
4.3 Uso della direttiva *ngFor con un array	54
4.4 Uso della direttiva *ngFor con un oggetto JSON	56
4.5 Creiamo il componente Treni dell'app MetroChat	58
4.6 Visualizzare o nascondere elementi del DOM: direttiva *ngIf	60
4.7 Esempio d'uso della direttiva *ngIf	62
4.8 Esempio d'uso del blocco else	64
4.9 Condizioni multiple: direttiva *ngSwitch	65

Capitolo 5 - Cambiare lo stile di elementi del DOM

5.1 Aggiungere o togliere proprietà CSS con ngStyle	69
5.2 Aggiungere o togliere regole CSS	72
5.3 Uso di ngClass con una proprietà stringa	72
5.4 Uso di ngClass con un array	73
5.5 Uso di ngClass con un oggetto	74

Capitolo 6 - Formattare i dati con i PIPE

6.1 Formattare le date e l'ora	75
6.2 Formattare stringhe e numeri	77
6.3 Creare Pipe personalizzati	79

Capitolo 7 - Modellare i dati

7.1 Una classe per definire il tipo di dati	83
7.2 Il modello dati per l'applicazione MetroChat	87
7.3 Una classe senza costruttore	88
7.4 Simulare dati remoti per semplici test	89
7.5 Classe con dati opzionali	90

Capitolo 8 - Interagire con il template e l'app: gli Eventi

8.1 Gestire il click o il tocco su un elemento del template	93
8.2 Gestire il click su un elemento del componente <ca-listnews>	95
8.3 Gestire gli eventi del mouse	97
8.4 Gestire gli eventi della tastiera	99

Capitolo 9 - Progettare Componenti “Intelligenti”

9.1 Cos’è un componente “intelligente”?	103
9.2 Proprietà di ingresso a un componente con @Input	105
9.3 Definire più proprietà di ingresso	109
9.4 Cambiare il nome alla proprietà di ingresso	110
9.5 Mostrare le informazioni, su un componente dettaglio interno	111
9.6 Definire proprietà di uscita con @Output()	114
9.7 Accedere a membri di altri componenti con @ViewChild	119
9.8 Accedere a membri di un componente figlio direttamente dal template	122

Capitolo 10 - La navigazione in Angular

10.1 Il concetto di Routing e Route	127
10.2 Configurare il file AppModule	129
10.3 Progettare l’array di route	130
10.4 Indicare dove visualizzare le route con RouterOutlet	132

10.5	Configurare i link nel template con RouterLink	134
10.6	Progettare Template indipendenti dal componente radice	135
10.7	Progettare route di Redirect	137
10.8	Progettare route di Pagina non trovata	139
10.9	Progettare route con figli	140
10.10	Impostare il Template della route di dettaglio per l'applicazione MetroChat	142
10.11	Progettare route con parametri dinamici	145
10.12	Recuperare i parametri da un url	150
10.13	Modificare il menu di dettaglio per l'applicazione MetroChat	152

Capitolo 11 - Separare le funzionalità con i Service

11.1	Perché usare i Service?	155
11.2	Come creare un Service	157
11.3	Registrare un Service	159
11.4	Come usare un Service	161
11.5	Gestire Dati Remoti con i Service	161

Capitolo 12 - Accedere a dati remoti

12.1	Manipolare dati via HTTP	165
12.2	Recuperare dati in GET	167
12.3	Inviare dati in modalità POST	176
12.4	Aggiornare dati in PUT	189
12.5	Cancellare i dati con DELETE	191

Capitolo 13 - Creazione e gestione di Moduli Web

13.1	I Form in Angular: introduzione	193
13.2	Cosa verificare prima di usare i Form	194
13.3	Creare il Formm con la tecnica “Template Driven”	196
13.4	Gestire i dati da un campo input	197
13.5	Accedere ad un campo di <input> tramite una variabile nel template	204
13.6	Settare un dato all'interno di un campo di input	208
13.7	Gestire i dati di un campo di select	209
13.8	Settare un valore all'interno di un campo select	211
13.9	Gestire i dati di un campo checkbox	212
13.10	Gestire i dati di un campo radiobox	215
13.11	Creare il Formm con la tecnica “Reactive”	216
13.12	Gestire i dati da un campo <input>	220
13.13	Recuperare il valore dei campi dopo il submit	224
13.14	Gestire i controlli dei campi di un form	226
13.15	Settare un dato all'interno di un campo di <input>	229
13.16	Gestire i dati di un campo di <select>	231
13.17	Settare un valore all'interno di un campo <select>	233
13.18	Gestire i dati di un campo <checkbox>	233
13.19	Gestire i dati di un campo radio	235

Capitolo 14 - Le basi della libreria RxJS

14.1 Dalla programmazione asincrona agli Observable	237
14.2 Peculiarità di un Observable	242
14.3 Tecniche per creare Observable	249
14.4 Cold vs Hot Observable	254
14.5 Operatori pipe(), map(), tap(), filter()	258
14.6 MultiCast Observable: Subject vs Behaviour Subject	268
14.7 Combinare flussi di Observable	281
14.8 Gestire gli errori	292

Capitolo 15 - Animare elementi di una vista

15.1 Introduzione al concetto di animazione	295
15.2 Installare la libreria per le animazioni	296
15.3 Definire i nomi delle applicazioni	298
15.4 Gli stati di un'animazione	3029
15.5 Definire le proprietà CSS di uno stato o animazione	304
15.6 Collegare l'animazione all'elemento nella vista	306
15.7 Il passaggio da uno stato all'altro	309

Capitolo 16 - Animare le transizioni tra pagine

16.1 Cosa si intende per transizione tra pagine	315
16.2 Configurare le route	316
16.3 Impostare il trigger a livello di contenitore pagina	317
16.4 Come definire l'animazione d'ingresso e uscita	320
16.5 Spostare il menu al di fuori del template di ogni componente	324

Capitolo 17 – Progressive Web App (PWA)

17.1 Passare da una Web App a una PWA	327
17.2 Cos'è un Service Worker e perché si usa	329
17.3 Usare Angular CLI per creare una PWA	331
17.4 Creare la versione di produzione e testare il Service Worker	333
17.5 Rendere l'app installabile: il file App Manifest	337
17.6 Configurare il Service Worker	341
17.7 Informare l'utente di un nuovo aggiornamento	345
17.8 Ciclo di vita di una PWA con Service Worker	347

Conclusione e Codice Personale	351
---------------------------------------	-----

APPENDICE	353
------------------	-----

Sitografia	357
-------------------	-----

Che cosa realizzeremo?

L'applicazione di fantasia che progetteremo in queste pagine, l'ho chiamata **"MetroChat"** e la puoi vedere completa di grafica a questo link:

<https://www.dcopelli.it/metrochat/>

Nella prima schermata, potrai visualizzare la lista dei treni in partenza presso una determinata stazione metropolitana, con la possibilità di selezionarne uno, ed entrare così in chat con gli utenti - di quel treno - che stanno chattando.

Per ogni treno, sarà possibile visualizzare un conto alla rovescia, che permetterà all'utente di conoscere il tempo mancante alla partenza.



Treni in Arrivo a questa stazione

Qui sotto puoi vedere l'elenco dei treni in arrivo presso questa ipotetica stazione. Cliccando sul simbolo del treno, potrai accedere al dettaglio, in cui vedrai la lista degli utenti che stanno chattando. Potrai inserire il tuo messaggio e aggiungere quello degli altri alla lista dei preferiti, in modo da ricontrattarli privatamente.

 Gialla Direzione: Zuld • 4 • C4	 00:29
 Verde Direzione: Isolatorweg • 3 • B1	 01:17

Treni appena partiti

Qui sotto puoi vedere la lista dei treni che sono appena partiti e che hai perso.

 Rossi Direzione: Centraal Station • 12 • A2
--

Figura 0.1 – Elenco treni in arrivo e appena partiti dalla stazione

Cliccando in corrispondenza ad un particolare treno, si entrerà nella schermata di dettaglio, in cui saranno riepilogati i dati del treno e visualizzati tutti i messaggi scambiati tra gli utenti di quel treno.

The screenshot shows a mobile application interface for a train chat. At the top, a blue header bar contains a back arrow icon and the word "Chat". Below this is a large title "Metro Chat". Underneath the title is a white card containing information about the train: "Rossa", "Direzione: Centraal Station", and "12 A2". Below the card is a search bar with the placeholder "Filtrai i messaggi". The main area displays a list of messages from different users, each with a profile picture, the message text, and a blue heart icon indicating it can be liked. The messages are:

- Oggi e' il mio primo giorno di lavoro e ho dimenticato la merenda! (blue heart)
- Se vuoi ti porto un panino con il salame! (red heart)
- Ciao Martina, piacere di rincontrarti. Come stai? (blue heart)
- Oppure un panino con il prosciutto (blue heart)

At the bottom, there is a text input field "Scrivi il tuo messaggio" and a send button represented by a black arrow pointing right.

Figura 0.2 – Elenco messaggi chat scambiati nel treno linea “Rossa”

Per ogni messaggio di chat, sarà possibile visualizzare l'utente che l'ha spedito, e inserirlo tra la lista dei preferiti, in modo da avviare una chat privata oltre che a filtrare i messaggi in funzione del testo contenuto.

Quest'ultima funzionalità sarà proposta come esercizio finale per ricevere un attestato di partecipazione.



Per evitare di appesantire la spiegazione, alcuni dettagli saranno omessi, come la rappresentazione grafica dei diversi elementi, che nella spiegazione sarà lasciata allo stato grezzo, così come l'identificazione dell'utente e della posizione da cui recuperare la lista dei treni o metro.

Il codice CSS e HTML completo dell'intera applicazione sarà comunque disponibile per il download dalla tua area privata, previa registrazione (vedi fino libro).

Inoltre, per il recupero dei dati, utilizzeremo una semplice API d'appoggio creata ad hoc, che non farà parte delle spiegazioni, ma che potrai sfruttare, per testare le funzionalità dell'applicazione (vedi appendice).

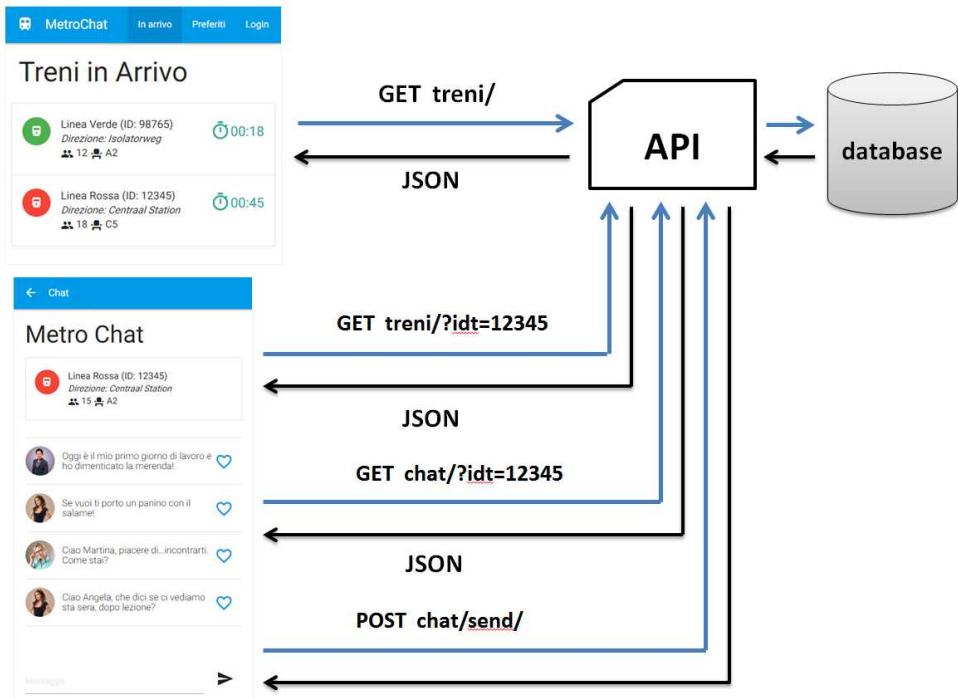


Figura 0.3 - Struttura chiamate API al server di database

Capitolo 1

Settaggio dell'ambiente di lavoro

1.1 Da dove partire: l'ambiente di test locale

Come per tutti i nuovi linguaggi di programmazione, che in 20 anni di attività sul web ho dovuto imparare, anche per il framework Angular, il primo passo da fare è dotarsi dell'ambiente su cui effettuare dei test.

Se per imparare il linguaggio HTML è sufficiente un browser e un software per scrivere del testo, per Angular, così come per altri framework basati su JavaScript è necessario installare Node.js e Npm.

Per ora non ci pre occuperemo di capire a cosa servono esattamente e che caratteristiche hanno, perché li useremo solo per installare nel computer una serie di strumenti utili alla creazione di nuovi progetti.

Agli inizi dello sviluppo di Angular, gran parte del lavoro di configurazione, doveva essere fatto manualmente.

Ora, con una sola riga di comando, creeremo un progetto completo da cui partire per iniziare a inserire tutte le modifiche del caso, un po' come se avessimo il classico file HTML di una pagina web, con già inseriti i tag <html><head><body> etc.

Impostare l'ambiente di sviluppo sul proprio PC

Come dicevamo, il primo passo è installare le librerie Node.js e Npm. E' un'operazione molto rapida. Tipicamente non sono presenti nei computer a livello predefinito.

1) Installare Node.js e Npm

Collegati al link del sito ufficiale di Node.js , in particolare alla sezione di download (<https://nodejs.org/en/download/>) e seleziona il sistema operativo su cui installerai i due software.

Il file che salverai sul tuo pc, contiene sia Node.js che Npm.

NB: Nel caso tu abbia già installato queste librerie in precedenza, accertati di avere delle versioni adeguate. Per Node.js deve essere superiore o uguale alla 18.10.x e per npm superiore alla 9.2.

Il test lo puoi fare lanciando le seguenti righe all'interno della classica finestra del prompt dei comandi, sia su Window, sia su Mac e Linux.

```
node -v // per verificare la versione di node.js installata  
npm -v // per verificare la versione di npm installata
```

2) Installare Angular CLI

Il passo successivo è quello di installare un insieme di librerie necessarie a eseguire dei comandi direttamente dalla finestra terminale.

Per dare un'occhiata all'attuale versione delle librerie e a eventuali operazioni da fare per aggiornare progetti Angular datati, collegati al link ufficiale su Github qui: <https://cli.angular.io/>

Dovrai allora scrivere questa riga:

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

NB: Fai attenzione a rispettare gli spazi presenti nella stringa e armati di pazienza, perché l'installazione potrebbe richiedere qualche minuto.

3) Creare un nuovo progetto Angular

Per creare l'insieme dei file necessari ad entrare nel vivo dello sviluppo della tua prima applicazione, puoi usufruire del comando qui sotto, che ti permetterà di creare un nuovo progetto dal nome *miapp*:

```
ng new miapp
```

Ti consiglio vivamente di creare prima una cartella di lavoro (es. TestAngular), e spostarti qui dentro per creare il progetto. Se non lo ricordi più, per cambiare la posizione usando la finestra terminale, dovrà usare i classici **comandi DOS**: *cd nomecartella* per entrare in una sottocartella, e *cd..* per spostarsi nella cartella padre.

```
C:\Users\videocorsi>cd Desktop
C:\Users\videocorsi\Desktop>cd TestAngular
C:\Users\videocorsi\Desktop\testAngular>ng new miapp[]
```

A seconda della versione di angular CLI (vedi punto 5), dovrà rispondere ad una serie di domande, tra cui se vuoi abilitare la cattura di errori per un migliore debug (rispondi di “Sì”), se vuoi abilitare il routing (rispondi “No” per ora) e che forma di foglio di stile usare (rispondi selezionando la voce “CSS”).

```
? Would you like to add Angular routing? No
? Which stylesheet format would you like to use? (Use arrow keys)
> CSS
SCSS [ https://sass-lang.com/documentation/syntax#scss ]
Sass [ https://sass-lang.com/documentation/syntax#the-indented-syntax ]
Less [ http://lesscss.org ]
Stylus [ http://stylus-lang.com ]
```

NB: Il nome da dare all'applicazione, deve essere scritto, senza inserire spazi, caratteri alfanumerici, simbolo di underscore (trattino basso).

In alternativa, puoi usare l'opzione *-S*, per limitare il numero di file che verranno installati nel progetto.

```
ng new miapp -S
```

Armati di pazienza, perch é questo passaggio potrebbe richiedere alcune decine di secondi, in quanto dovranno essere scaricati diversi pacchetti *npm*.

4) Avviare il server e caricare la pagina nel browser

Arrivati a questo punto, nella cartella in cui hai creato il progetto, troverai la sottocartella *src*, che conterrà tutta una serie di file, suddivisi tra file di configurazione e file dell'app, che impareremo a capire in dettaglio nelle prossime lezioni.

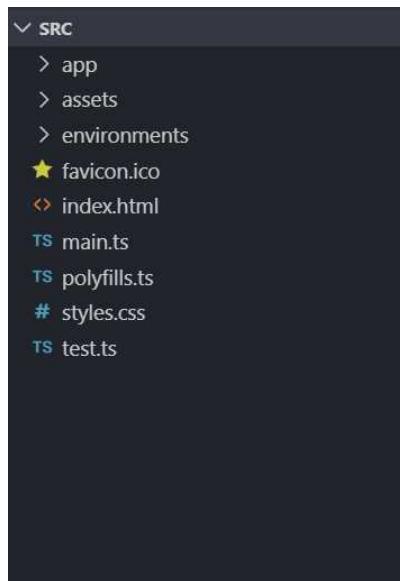


Figura 1.1 – Elenco file cartella *src* di un progetto Angular

Purtroppo non è possibile lanciare l'applicazione cliccando su uno specifico file *index.html*, come avviene per le classiche pagine web, ma dovrà prima di tutto avviare un server locale integrato, che ti permetterà di aprire l'applicazione da un indirizzo web specifico, in particolare dall'indirizzo <http://localhost:4200/>

NB: Fai attenzione a inserire la porta ossia il numero :4200/

Sempre con l'ausilio della finestra terminale, dovrà spostarti all'interno della cartella in cui è stato creato il progetto, nel nostro caso all'interno della cartella *miapp*:

```
cd miapp
```

e poi avviare il server di produzione locale, che ci servirà per interpretare correttamente le pagine:

```
ng serve --open
```

NB: Ricordati che devi sempre essere all'interno di una cartella in cui è presente un progetto Angular. L'opzione `--open` (occhio ai due simboli di "meno"), serve per aprire in automatico la finestra del browser predefinito e di ricaricarla in automatico ad ogni modifica del codice. Ti consiglio di usare Chrome per tutti i test.

Grazie a questo comando, ogni modifica che introdurremo ai diversi file dell'app, automaticamente si rifletterà nella pagina visualizzata dal browser.

NB: Spesso può capitare che le ultime versioni di `angular-cli` non funzionino correttamente, quindi nel caso la pagina non dovesse aggiornarsi, dovrà armarti di pazienza e attendere che sia corretto il bug e aspettare la nuova versione o installare quelle precedenti.

Bene, se il browser non dovesse aprirsi, dovrà collegarti al link qui sotto:

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

per visualizzare il messaggio "Miapp is running !" (o simile), che ti confermerà la correttezza di tutti i passaggi fatti fino ad ora per configurare l'ambiente di sviluppo in locale.

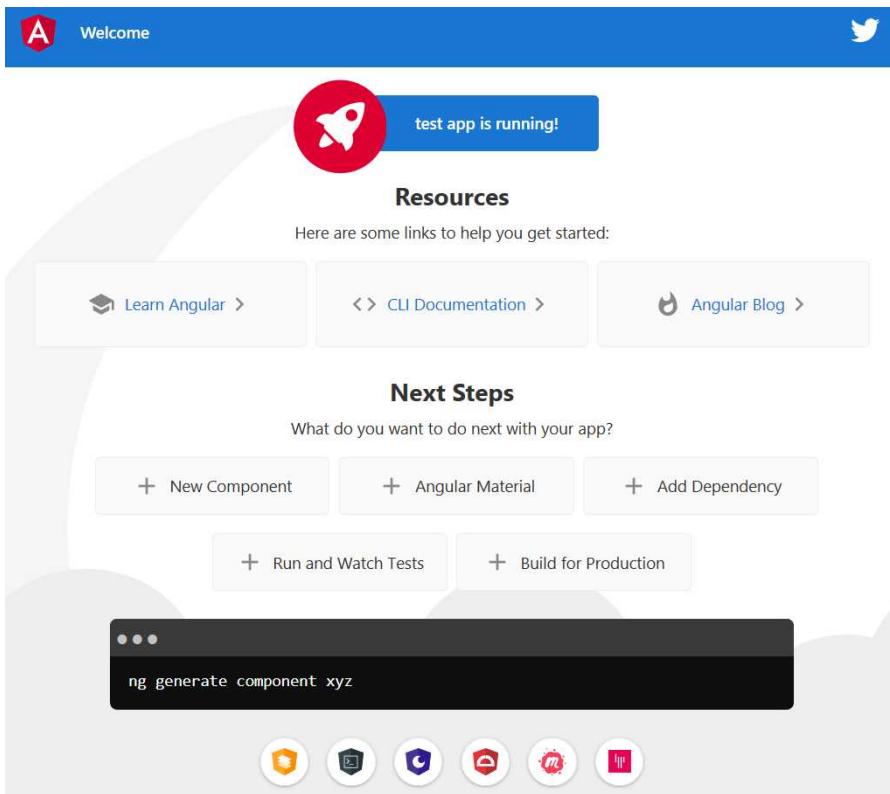


Figura 1.2 – Schermata iniziale applicazione Angular, visualizzate nel browser

Il file che viene visualizzato dal browser, è proprio il file *index.html*, presente all'interno della cartella radice del tuo progetto (cartella *src*). Se tu sbirciassi all'interno del codice sorgente di questa pagina, non troveresti granché: solo tag HTML.

Andando però a guardare con lo strumento "ispeziona documento" del browser, troveresti anche una serie di file JavaScript, iniettati nel codice della pagina in fase di compilazione.

In realtà, vedremo che lo "strano" tag `<app-root>` presente all'interno della pagina, non è un classico tag HTML, ma un “potenziamento” del codice HTML.

Non appena il browser carica la pagina, Angular trasforma a questo tag, in modo che sia correttamente interpretato, sulla base d'informazioni che sarà nostra cura inserire all'interno dell'applicazione.

5) Configurare il compilatore

Quando si crea un nuovo progetto usando le ultime versioni di Angular, ti potrebbe venire richiesto di scegliere la modalità *strict*, quindi ricevere la segnalazione di tutti gli errori TypeScript, anche quelli che alcuni sviluppatori ritengono superflui o non propriamente errori.

Uso il condizionale perché è un settaggio che ho visto cambiare spesso da una versione all'altra di Angular. Puoi verificare la modalità attualmente in uso, aprendo il file *tsconfig.json*, che trovi nella cartella radice del progetto che hai appena creato. Se in corrispondenza alla voce “*compilerOptions*” trovi la chiave:

```
"strict": true
```

significa che tutte le segnalazioni di errori Typescript sono attive.

Il mio consiglio da “hacker” è quello di disabilitare una parte di queste segnalazioni durante la fase di sviluppo, per poi abilitarle nella fase di test finale, giusto per verificare se alcune meritano di essere prese in considerazione.

Spostati quindi nel file *tsconfig.json*, e all'interno di “*angularCompilerOptions*” aggiungi o verifica ci siano le seguenti righe:

```
"angularCompilerOptions": {
  ...
  "strictPropertyInitialization": false, // impostala a false
  "noImplicitAny": false, // impostala a false
  "suppressImplicitAnyIndexErrors": true // impostala a true
}
```

Ad ogni modifica di questo file, dovrai terminare l'esecuzione del server e riavviarlo (vedi punto successivo).

6) Terminare l'esecuzione del server

Non appena avrai concluso tutti i test sulla tua applicazione, conviene arrestare il sever locale avviato in precedenza con *ng serve*. Questo per non rallentare il proprio PC con programmi che non servono più. Analogico discorso quando applichi delle modifiche ai file di configurazione, come visto in precedenza.

Per farlo, dovrai chiudere la finestra terminale oppure digitare la combinazione di tasti CTRL+C.

E' chiaro che per poter nuovamente testare l'app, dovrà aprire la finestra terminale, spostarsi nella cartella del tuo progetto e digitare nuovamente la riga vista nel punto 4.

Abbiamo imparato i pochi passaggi necessari per preparare il proprio computer all'esecuzione e test di un'applicazione creata con il framework Angular. Questi passaggi li dovrà ripetere per ogni nuova applicazione.

1.2 I migliori IDE per scrivere applicazioni

Dopo aver configurato il tuo computer al fine di poter testare la prima applicazione Angular, il passo successivo è quello di scegliere quale "editor" utilizzare per scrivere il codice.

Qui l'offerta non manca mai, come sempre, la scelta deve cadere sul migliore, e con migliore intendo quello strumento in grado di aiutarti a scrivere meno codice possibile, grazie all'auto-completamento di questo, a eventuali "snippets" e ad aiuti visivi, come la colorazione dei diversi elementi del linguaggio.

Il linguaggio preferenziale per scrivere applicazioni Angular è TypeScript.

Ecco allora la mia personale classifica dei migliori editor di testo, gratuiti e non, per scrivere applicazioni Angular con TypeScript.

Visual Studio Code - Gratuito

Per molti anni, la Microsoft, ha offerto un software a pagamento, il famoso "Visual Studio", rivolto principalmente agli sviluppatori di applicazioni ASP.net. Poi è passata alla versione gratuita, con "Visual Web Developer", e infine ha creato "Visual Studio Code".

A mio avviso è il software più adatto per la scrittura di codice, grazie alla presenza della funzionalità di "IntelliSense" ereditata da Visual Studio. Lo puoi scaricare da questo link: <https://code.visualstudio.com/>

Altra funzionalità interessante, è la possibilità di aggiungere delle "estensioni" in grado di potenziarne il funzionamento. L'icona da cliccare è quella

Ti consiglio di installare, o quella sviluppata da John Papa o da Mikael Mörlund. Ti permetteranno di risparmiare molto tempo nella scrittura di pezzi di codice Angular, grazie all'inserimento di snippet di codice.

Le trovi cercando la voce "Angular Snippet". Fai attenzione che non sempre le estensione sono aggiornate all'ultima versione di Angular, anche se spesso le differenze tra una versione e l'altro sono minime.

Chiaramente ne esistono decine e dovrà scegliere quella che più si adatta al tuo stile. Per installarla, dovrà cliccare sull' icona "Estensioni" presente nel la barra laterale sinistra, oppure digitare contemporaneamente i tasti **CTRL+Maiuscolo+X** e nella barra di ricerca, digitare il tipo di estensione.

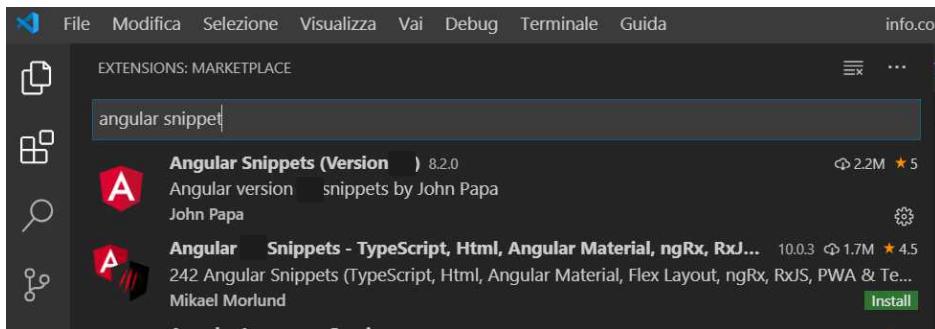


Figura 1.3 – Elenco estensioni Visual Studio per velocizzare l'inserimento del codice

Una volta installata, ad esempio quella di John Papa , l'uso è molto semplice, perché sarà sufficiente aprire un file TypeScript e iniziare a digitare la sigla `a-`, per vedere caricare una finestra con tutta una lista di nomi, legati a particolari pezzi di codice (snippets), che frequentemente dovrà inserire in un'applicazione Angular e che saranno oggetto del corso.

Ti invito a fare qualche prova per vedere cosa viene inserito nella pagina.

Altra estensione importante è quella che ti permette di auto completare il codice, con l'importazione delle diverse librerie.

Questa estensione è particolarmente utile all'inizio, quando ancora non si ha dimestichezza con i diversi package presenti in Angular.

L'estensione si chiama "AutoImport" e una volta installata, ti permetterà di aggiungere in automatico le diverse librerie non appena aggiungerai un oggetto, un componente etc.

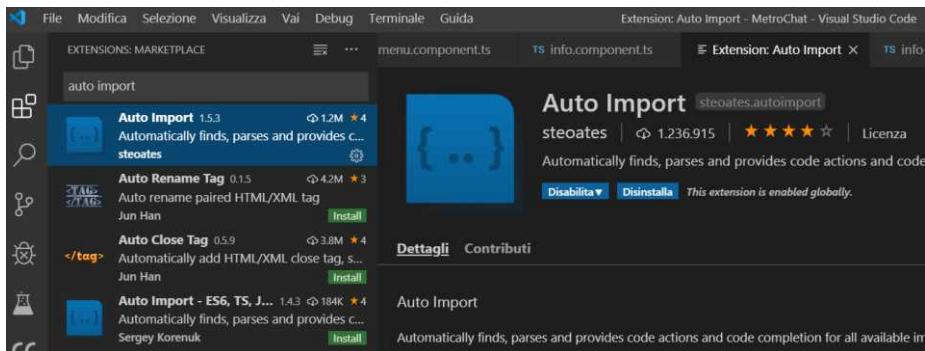


Figura 1.4 – Estensione Auto Import per importare i moduli Angular automaticamente

ATOM - Gratuito

L'editor “Atom”, da molti anni è considerato uno dei migliori software open source per la scrittura di programmi, compatibile con diverse tipologie di linguaggio, grazie alle decine di plugin.

Il software lo puoi scaricare da questo link: <https://atom.io/>. Dopo averlo scaricato, devi ricordarti di installare anche il plugin "TypeScript Plugin".

L'installazione di un plugin è un passaggio molto semplice e veloce, perch è dovrà solo accedere al menu "File>Settings" e poi selezionare , nella colonna centrale alla pagina, la voce "Install" per accedere ad una finestra simile a quella qui sotto, da dove potrai cercare i diversi plugin da installare.

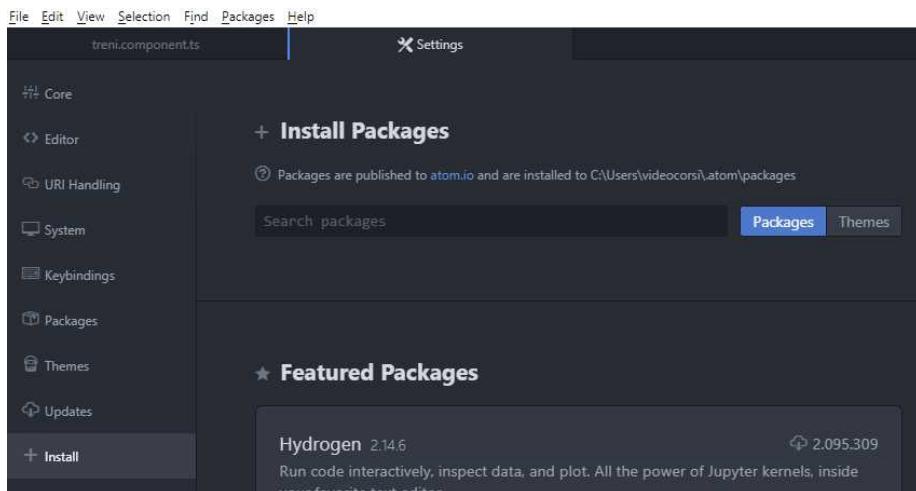


Figura 1.5 – Installazione di un Plugin in Atom

Sublime Text - A pagamento

Quest'ultimo è un software che tutti i possessori di Mac, conoscono molto bene. Esiste tuttavia anche la versione adattata per windows. Lo puoi scaricare da questo link: <http://www.sublimetext.com/>. Oltre al software, dovrà ricordarti di installare anche il plugin "TypeScript-Sublime"

WebStorm - A pagamento

Altro software degno di nota, ma ahimè anch'esso a pagamento, è "WebStorm". Puoi scaricarne una versione demo di 30 gg da questo link:
<https://www.jetbrains.com/webstorm/download>

1.3 Comandi Utili CLI

Abbiamo visto che lo strumento per creare un nuovo progetto è Angular CLI accessibile tramite il prompt dei comandi.

Qui sotto puoi trovare una lista di azioni utili per creare i diversi elementi che compongono un'applicazione Angular e che ti permetteranno di risparmiare tempo nella scrittura del codice.

Le scritte in maiuscolo, sono quelle che dovrà personalizzare.

a) Creare una nuova App Angular

```
ng new NOME // o in alternativa ng new NOME --minimal
```

b) Testare l'applicazione in locale

Il comando deve essere eseguito spostandosi nella cartella del progetto:

```
ng serve --open
```

c) Creare nuovi elementi dell'App

Per creare i diversi elementi, dovrai sempre spostarti nella cartella in cui vuoi che siano creati e lanciare il comando `ng generate` (o l'alias `ng g`), seguito dal tipo di elemento e dal NOME da assegnargli:

Componente	<code>ng g component NOME</code>
Direttiva	<code>ng g directive NOME</code>
Pipe	<code>ng g pipe NOME</code>
Service	<code>ng g service NOME</code>
Class	<code>ng g class NOME</code>
Interfaccia	<code>ng g interface NOME</code>
Modulo	<code>ng g module NOME</code>

d) Aggiornare un progetto alle ultime versioni di Angular

Una delle caratteristiche più “stressanti” per chi deve imparare una nuova tecnologia è il continuo e inevitabile aggiornamento delle versioni. Angular amplifica questo stress, in quanto viene aggiornato a cadenza mensile.

E’ importante quindi sapere come aggiornare il proprio ambiente di test per rimanere al passo con il rilascio delle nuove versioni.

Se ti iscriverai alla “ WEB StartUniversity (WEBSU)”, potrai rimanere aggiornato con tutti i codici che svilupperemo nel libro, oltre ai diversi tutori online e corsi che svilupperemo nel tempo.

Il processo di aggiornamento varia a seconda della versione in cui è stato sviluppato il progetto di partenza.

Collegandoti a questo indirizzo: <https://update.angular.io/>, potrai ricevere istruzioni dettagliate sui diversi passi da seguire per aggiornare il tuo progetto.

Se provieni da vecchie versioni <7.x, il punto di partenza è aggiornare Angular CLI, sia in locale al progetto che a livello globale, e cambiare la struttura del file angular.json

Apri la finestra terminale, entra nella cartella del tuo progetto e scrivi, una alla volta, le righe seguenti, schiacciando il tasto invio al termine dell'inserimento di ognuna:

```
ng update @angular/cli
ng update @angular/core
```

Nel caso dovessero comparire degli errori puoi provare la riga:

```
ng update --all --force          // solo in casi estremi !!!
```

Ricordati che Angular si aggiorna ogni 5/6 mesi, quindi se vuoi mantenere il tuo progetto aggiornato, ricordati di rieseguire questi passaggi. Per verificare la versione attualmente installata sul tuo PC, sia di Angular CLI sia di Angular, digita:

```
ng version
```

e) Creare la versione di produzione da pubblicare nel proprio sito

Per progetti di bassa complessità come il nostro, sarà sufficiente scrivere la riga qui sotto. Nella cartella radice del progetto, comparirà una nuova cartella con nome “*dist*”. Tutti i file presenti al suo interno, dovranno essere trasferiti nel proprio sito web.

```
ng build --prod
```

Fai attenzione che se decidi di pubblicarlo in una sottocartella del tuo dominio, ad esempio <http://www.pincopalla.it/angularapp/>, dovrà modificare il file `index.html` in modo che il meta tag `<base>` tenga conto della nuova posizione (Es. `<base href="/angularapp/">`).

Capitolo 2

Il motore di un'app Angular

2.1 La struttura di un'App Angular

Per creare un'app in Angular è necessario avere un po' di dimestichezza con JavaScript e i concetti della programmazione ad oggetti, quindi è consigliabile aver seguito un corso che tratta di JavaScript.

Questo perché? Perché il linguaggio da preferire per lo sviluppo, è TypeScript, che è un'estensione di JavaScript creata da Microsoft.

Infatti, il cuore di ogni app Angular, è costituito da quelli che vengono chiamati "componenti", ossia un insieme di tag, estensione del codice HTML, con cui riesco a realizzare ogni pagina della mia applicazione.

I componenti se vogliamo, sono un'**evoluzione** del linguaggio HTML e stanno diventando sempre più importanti in molti framework (vedi Polymer, React, etc), poiché ora la maggior parte dei browser è in grado di supportarli.

Per maggiori dettagli, visita la pagina dedicata: WebComponents.org

Se ad esempio, prendessi in esame l'interfaccia di fig.2.1, potrei dire che è costituita da un certo numero di componenti: da i pulsanti del menu, dalla singola ricetta, dai pulsanti “favoriti” e così via.

Ognuno di questi può contenere all'interno altri componenti, legati tra di loro con una relazione “genitore/figlio” o “padre/figlio”.

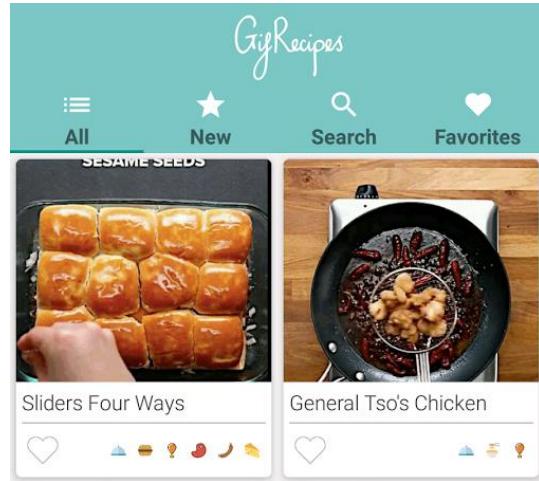


Figura 2.1 – Tipica interfaccia utente di un’App

In Angular, ogni componente sarà caratterizzato da tre **elementi fondamentali**:

1. il **nome (selettore)**
2. la sua rappresentazione **grafica (template o vista)**
3. le funzionalità che deve avere (**classe o modello**)

Un'applicazione, si può paragonare all'astronave “Apollo 13”, che come si vede nell'omonimo film, era costituita da più moduli, tra cui il lem, quello che gli astronauti usarono per ricongiungersi con il modulo in orbita per poi ritornare a terra.

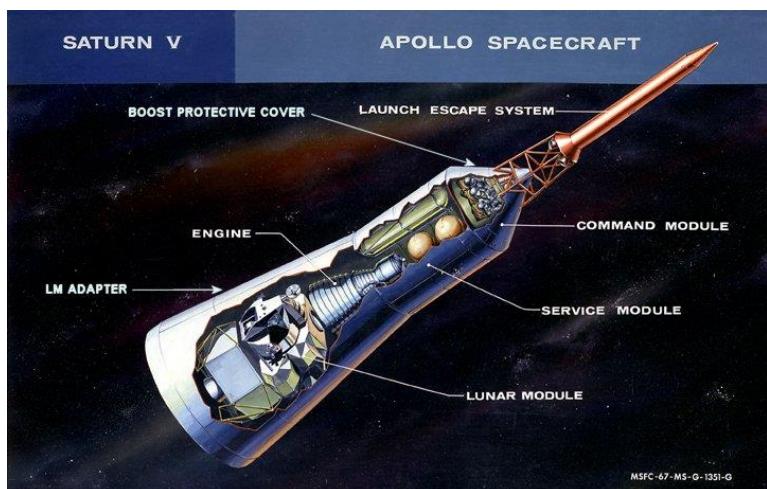


Figura 2.2 – Apollo Spacecraft Wikipedia

La rappresentazione grafica è sicuramente la parte meno complessa, in quanto basta che tu conosca la sintassi del codice HTML, o tu sappia già usare un framework responsive, o Material Angular o Ionic, (vedi i miei corsi WebSU) e il gioco è fatto.

L'unica cosa da approfondire, è capire come sia possibile iniettare dei valori, dal corpo del componente o classe, al template e viceversa. Di tutto questo parleremo in una sezione specifica del libro.

Chiarito che un'app Angular è costituita da tanti componenti e da alcuni moduli, la domanda che potresti farti è: ma come si **crea una pagina** per un'applicazione Angular? Ad esempio come posso creare l'ipotetica "homepage" della mia app?

Ebbene, come ogni razzo spaziale, che contiene tutte le attrezature per la spedizione nello spazio, anche ogni applicazione Angular, deve essere progettata partendo da un file iniziale, che raggrupperà tutto il necessario per far partire l'applicazione, una sorta di equivalente pagina di ingresso di un sito web.

Per convenzione, questa pagina dovrà avere nome, non Apollo 13 ma `index.html`, e si progetterà come una semplice pagine web, con la differenza che al suo interno dovrà inserire sia i classici tag del linguaggio HTML, che quelli legati ai componenti da me progettati.

Il passo successivo sarà quello di far “partire il razzo”, quindi avrai bisogno di un “sistema di propulsione” (modulo radice) e di una “miccia” (file `main.ts`)

Riassumendo, per far partire un'applicazione, sono necessari tre elementi:

1. Un file `index.html`, con interno almeno un componente.
2. Un Modulo (file `app.module.ts`), equipaggiato con tutti gli elementi che servono all'applicazione per la compilazione e il lancio.
3. Una "miccia" (file `main.ts`), ossia un file che faccia partire l'applicazione.

Come vedi, rispetto a un semplice sito web - in cui basta creare una pagina HTML e aprirla con un qualsiasi browser - il passo iniziale è più complesso e richiede diverse conoscenze, che cercheremo di approfondire nelle prossime pagine.

2.2 Il file index.html

La struttura della pagina principal e, come dicevamo, si crea con le stesse tecniche con cui si progetta una classica pagina html:

```

Index.html

<!DOCTYPE html>
<head>
  // importazione di una serie di librerie Angular
</head>
<body>
  <!--componente Angular-->
  <ca-miomenu></ca-miomenu>
  <h1>Articoli Angular</h1>
  <!--componente Angular-->
  <ca-listaarticoli></ca-listaarticoli>
</body>
</html>
```

dove i tag html `<ca-listaarticoli>` e `<ca-miomenu>`, sono i nomi che sceglierò per identificare dei futuri **componenti Angular** da progettare.

NB: Il nome del componente può essere scelto a piacere, anche se per convenzione si preferisce anteporre una sigla per distinguere i propri componenti da quelli sviluppati da altri. Nel nostro caso, abbiamo scelto la sigla ca-, che è l'iniziale del nostro sito web creareapp.com

NB: Non usare caratteri alfanumerici (&%\$ etc.) e lo spazio. Inserisci sempre tutte lettere minuscole.

Come dovranno essere progettati questi ultimi?

Prima di tutto è necessario conoscere la sintassi per creare una classe in TypeScript e poi è necessario conoscere la sintassi con cui si creano i componenti in Angular.

Questo però non è sufficiente, in quanto devi anche importare una serie di librerie necessarie al corretto funzionamento, altrimenti il browser non sarà mai in grado di capire cosa sostituire al posto di `<ca-miomenu>` o di `<ca-listaarticoli>`.

Non avere fretta, impareremo a fare tutte queste cose molto presto. Per ora concentriamoci sui passi per far partire la nostra prima applicazione.

Dove e come si crea questo file?

Non appena sfrutti i comandi CLI visti nel primo capitolo "Settings dell'Ambiente di Lavoro", in automatico si genera questo file, che sarà presente all'interno della cartella `src` del progetto.

File indice con Librerie ANGULAR caricate da SystemJS

Al fine di poter elaborare i nuovi tag html inseriti e capire come eseguire eventuali azioni "nascoste" nella logica di ciascun componente, è necessario aggiungere all'interno della cartella principale del progetto, una serie di file di configurazione, che fortunatamente sono aggiunti in automatico non appena creiamo un nuovo progetto, sfruttando i comandi in linea visti in precedenza.

Fino alla versione beta.10 di angular -cli, il sistema usato per caricare una serie di "pezzi" necessari al funzionamento dell'applicazione, si basavano sulla libreria "SystemJS". Quest'ultima è una libreria che ha lo scopo di **caricare dei moduli**, ossia dei file che conterranno all'interno altri pezzi dell'applicazione oltre che una serie di altri file collegati.

Era tipico quindi avere dei file indice come questo:

```

Index.html

<!DOCTYPE html>
<head>
  <title>Angular App</title>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <!-- importazione di una serie di librerie angular -->
  <!-- Polyfill(s) for older browsers -->
  <script src="node_modules/core-js/client/shim.min.js"></script>
  <script src="node_modules/zone.js/dist/zone.js"></script>
  <script src="node_modules/systemjs/dist/system.src.js"></script>
  <!--Librerie per caricare il primo Modulo dell'app-->
  <script src="systemjs.config.js"></script>
  <script>
    System.import('app').catch(function(err) { console.error(err); });
  </script>
</head>
<body>
  <app-root></app-root>
</body>
</html>
```

File indice con Librerie ANGULAR caricate da WebPack

Da diversi anni, il team di Angular, ha abbandonato SystemJS a favore della libreria “WebPack”, un sistema di caricamento dei moduli dell'applicazione Angular, decisamente più snello e agevole da usare.

Il primo beneficio, lo osserviamo subito guardando il codice sottostante, in cui ci si accorge al volo, che sono completamente sparite diverse righe interne a *meta tag* `<head>`.

```
index.html

<!DOCTYPE html>
<head>
  <title>Angular App</title>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width,initial-scale=1">
</head>
<body>
  <app-root>Loading</app-root>
</body>
</html>
```

Un passo avanti sicuramente verso una semplificazione rispetto al vecchio sistema.

2.3 Convenzioni per la progettazione del file `index.html`

I più attenti avranno notato che, il file `index.html` creato tramite la linea di comando, presenta all'interno un tag `<app-root>`, che ha le sembianze di un classico tag HTML, ma non corrisponde ad alcuno dei tag di tua conoscenza.

Si tratta del componente chiamato “**componente radice**”, e vedremo sarà uno degli elementi chiave di un'applicazione.

```
<app-root>Loading</app-root>
```

Una delle caratteristiche di Angular, è la capacità di tenere separati i diversi pezzi di un'app, per rendere la fase di sviluppo e di test, semplificata.

Pertanto, non avremo mai una situazione come quella rappresentata nel codice visto inizialmente, in cui nel file `index.html` erano presenti più componenti, ma si partira sempre dal codice simile a quello indicato qui sotto , in cui si intravede un **unico**

componente, identificato dal tag `<app-root>`, il quale a sua volta conterrà altri elementi HTML o altri componenti, con una struttura tipica ad albero rovesciato.

```

index.html

<!DOCTYPE html>
<head>
  <title>Angular App</title>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1">
</head>
<body>
  <app-root>Loading</app-root>
</body>
</html>

```

NB: Tutte le applicazioni Angular che progetterai, saranno caratterizzata dall'avere un solo componente nel file `index.html`.

Cosa serve ad Angular oltre al file `index.html`?

Come detto in precedenza, così come accade per un razzo che ha bisogno di una serie di elementi per partire (propellente, base di lancio, equipaggio, materiale etc.), anche per un'applicazione Angular, dobbiamo avere un meccanismo che ci consenta di mostrare il componente principale inserito nel file `index.html`.

Abbiamo bisogno di quello che si chiama in gergo tecnico “Modulo”, che conterrà all'interno tutto l'occorrente per far funzionare l'applicazione.

Questo modulo è chiamato **“Root Module”** per distinguerlo da eventuali altri moduli detti **“Feature Module”**, di cui sarà costituita un'applicazione più strutturata.

Vediamo allora come possa essere strutturato un file di questo tipo e dove debba essere creato.

2.4 Il mio primo Modulo: `AppModule`

Se nel mondo dello sviluppo di siti web in HTML le pagine html sono il fulcro dell'intero progetto, nel mondo dello sviluppo di applicativi Angular, la pagina `index.html` è l'equivalente dell'homepage.

La grossa differenza rispetto allo sviluppo di siti web, è che dobbiamo includere una serie di funzionalità che aiuteranno gli attuali browser a interpretare correttamente i

diversi elementi tipici di un'applicazione Angular, come i componenti , le direttive, i servizi.

In sostanza d ovremo predisporre una sorta di “cabina di regia ”, che includa queste informazioni.

Non prendere paura perché si tratta d 'inserire solo due file. Il primo è una semplice classe, “adornata” con il decoratore `@NgModule()` (vedremo cos'è un decoratore prossimamente).

Questa tipologia di classe , dicevamo in precedenza, è chiamata **Modulo**, e un'applicazione Angular avrà sempre un modulo principale, chiamato "**Modulo Radice**" o "AppModule".

Questo concetto potrebbe essere difficile da “digerire” e da ricordare, ma vedrai che, dopo aver visto i primi esempi, assimilerai questa terminologia rapidamente.

In precedenza, avevamo detto che il modulo radice “AppModule”, lo potevamo paragonare al motore di un razzo, che inizialmente è spento ma che, grazie alla “miccia” costituita dal file `main.ts`, sarà in grado di accendersi e recuperare quelle risorse necessarie al funzionamento dell'app.

Qui sotto puoi vedere un mio schizzo sulla lavagna, durante una lezione.



Figura 2.3 – Una mia lezione in aula

Dove è salvato il primo modulo e con che estensione?

Il file è salvato all'interno di una sottocartella `con` un nome facile da ricordare: `app` (scritta in minuscolo). Qui dentro, vedremo, salve remo molti altri file, sempre con una struttura a sottocartelle. L'estensione che dovrà avere sarà `.ts`, che è l'estensione di ogni file TypeScript. Il tipico nome usato per il modulo radice è `app.module.ts` e questo è il motivo per cui spesso si chiama anche “ `AppModule`”.

NB: Nota la presenza del punto di separazione tra la parola `app` e la parola `module`. Questa convenzione, sarà usata anche per altri file (es. i componenti).

La scelta di sfruttare questo meccanismo centralizzato per il caricamento delle risorse necessarie all'applicazione, è uno dei punti di forza di Angular.

Come deve essere creato questo file?

Il metodo consigliato è quello di sfruttare la linea di comando, perch é non appena si crea un nuovo progetto, sarà automaticamente creato anche questo file. Per ora non è necessario ricordare a memoria la struttura delle informazioni da inserire all'interno ma solo cosa deve essere inserito al fine di far funzionare l'applicazione.

Sintassi del file `app.module.ts`

Essendo una classe, la tipica rappresentazione del modulo radice, avrà una struttura di questo tipo:

`app/app.module.ts`

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

import { MiaAppComponent } from './app.component';
import { MenuComponent } from './menu/menu.component';

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

Assimilerai l'uso e il significato di ognuna, durante i vari esempi che vedremo durante il percorso di creazione dell'applicazione MetroChat. Ecco una rapida spiegazione:

- Il modulo, è chiamato per convenzione “`AppModule`”, ed è salvato con il nome `app.module.ts` all'interno della cartella `app`.
- Ci deve essere sempre un solo “modulo radice”, per applicazione.
- Il modulo radice, deve sempre importare “`BrowserModule`”, che conterrà una serie di elementi importanti (direttive) come `*ngIf`, `*ngFor`. Vedremo che esistono altri moduli importanti come “`FormsModule`” e “`HttpModule`”
- All'interno della proprietà `declarations`, inserirai un array con i nomi di tutti i **componenti** usati nell'applicazione.
- All'interno della proprietà `bootstrap`, inserirai il nome del componente che dovrà essere **caricato per primo**.

Sintassi del file `main.ts`

Nell'analogia con il razzo vista in precedenza, per far partire l'applicazione Angular, abbiamo bisogno di una “miccia”, rappresentata dal file `main.ts`, anch'esso inserito durante il processo di creazione di un progetto con la linea di comando. Si trova all'interno della cartella `src`, insieme al file `index.html`.

Questo è il secondo file che ci serve. La tipica struttura sarà di questo tipo:

main.ts

```
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
import { AppModule } from './app/app.module';
platformBrowserDynamic().bootstrapModule(AppModule)
  .catch(err => console.error(err));
```

Non mi soffermo più di tanto sulla struttura, perché nella quasi totalità delle applicazioni che progetterai, non dovrà mai modificarla. L'unica cosa da osservare è la presenza della riga con il metodo `bootstrapModule()` che, come avrai capito, ha il compito di richiamare proprio il modulo creato in precedenza.

Il motivo per cui si è scelto di creare un file adibito al caricamento del Modulo principale dell'app, è legato al fatto che Angular è un framework che permette lo sviluppo di applicazioni in diversi ambienti, quindi non solo sul browser.

Cambiando la riga di lancio dell'applicazione, facilmente la potrai eseguire anche su ambienti diversi.

Capitolo 3

I Componenti in Angular

3.1 Il “cuore” di un’applicazione Angular

Come ricordato più volte, i 1 “cuore” di ogni applicazione Angular, è sicuramente costituito dai componenti. In questo capitolo cercheremo di capire come si creano , che caratteristiche hanno e come si inseriscono all’interno di una “vista”. E’ sufficiente infatti paragonare un’applicazione Angular alla squadra di calcio della tua città o alla classe di una scuola per convincersi dell’importanza di questo concetto.

Ogni giocatore, ogni alunno della classe, non è altro che un componente della squadra/classe, dotato di un proprio nome, della propria maglia e di peculiarità che lo contraddistinguono da tutti gli altri.



Figura 3.1 – I componenti di una squadra nell’analogia con i componenti di un’app

Se provi a guardare l'immagine dell'interfaccia di semplici APP come quelle di fig.3.1, ti convincerai che anch'ess a può essere pensata costituita da tanti elementi, ognuno dei quali ha uno specifico comportamento.

Business

Dow in 4-day win streak, hits new closing high
USA TODAY – 14 mins ago

At the end of the trading day, the Dow was up 0.5% to 17,138.20, a new all-time closing high. The blue-chip index topped its previous record close of 17,068.26 set on July 3. News of a takeover bid by Rupert Murdoch's 21st Century Fox for Time Warner ...

Wall St. gains on M&A, results; Dow ends at record high – Reuters – 25 mins ago

Merger Possibilities and Earnings Inspire the Market – New York Times – 1 hour ago

Opinion: Wall St. gains, Dow ends at record high – The Star Online – 1 hour ago

In-Depth: STOCKS RALLY, DOW MAKES RECORD HIGH: Here's what you... – SFGate – 46 mins ago

Show less ^

← Los Angeles, CA

AUGUST 18 – 23
5 NIGHTS • 1 ROOM • 2 GUESTS

Unlock Private Deals
Save up to 35% on hotels UNLOCK

	Sofitel Los Angeles at Beverly ...	\$313
★★★★★ Excellent 8.5 (2,685 reviews) Top Business Hotel		
	DoubleTree by Hilton Hotel Lo...	\$299
★★★★★ Good 8.0 (1,362 reviews)		
	Millennium Biltmore Hotel Lo...	\$236
★★★★★ Good 7.5 (2,419 reviews)		
	Sheraton Gateway Los Angele...	\$175
★★★★★ Good 8.0 (7,458 reviews)		
	The Moment Hotel	\$233
\$ (USD) per night, excluding taxes & fees		

Figura 3.2 – Suddivisione degli elementi di un' interfaccia Utente (UI) di un'applicazione

Per entrambe possiamo individuare il menu dell'intera applicazione e i singoli articoli. Ebbene in Angular, è possibile isolare ciascun elemento rappresentativo dell'interfaccia utente (UI) di un'APP, e trasformarlo appunto in un **Componente**.

Quest'ultimo quindi rappresenta un pezzo di quello che vedrà l'utente nel proprio schermo, e ognuno di questi pezzi è dotato di caratteristiche univoche e della possibilità di interfacciarsi con altri elementi dell'interfaccia utente.

Un componente inoltre, può avere all'interno altri componenti, detti componenti "figlio" (Child Component) e con cui il componente "genitore" o "padre" è in grado di comunicare.

Come già visto nel capitolo precedente, un componente deve avere allora **tre caratteristiche** fondamentali:

1. Deve avere un **nome identificativo (nome o selettore o tag)**
2. Deve avere una **rappresentazione grafica (template o vista)**
3. Deve avere delle **funzionalità (classe o modello)**

Applicando questi concetti alla prima applicazione di fig.3.1, potremmo dire che il menu è:

1. identificabile con un nome es. "ca-menu"
2. ha una rappresentazione grafica con del codice HTML che ne definisce la struttura e il testo da visualizzare
3. ha delle funzionalità tali da poter cambiare pagina non appena il navigatore tocca un link.

A livello tecnico, vedremo che un componente sarà rappresentato da un file con estensione *.ts*, e sarà a tutti gli effetti una **classe**.

3.2 Il componente <app-root> creato in automatico

Se hai già provato a creare la tua prima applicazione Angular con la linea di comando, avrà notato che, all'interno della cartella *app*, vengono aggiunti una serie di file, compreso quello con nome *app.component.ts*.

Ebbene questo file è proprio il primo componente visualizzato da Angular, perché è quello indicato all'interno del file *app.module.ts*, in corrispondenza alla proprietà *bootstrap*.

app/app.module.ts

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

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

export class AppModule {}
```

Il file con nome `app.component.ts`, definisce il componente con selettore `<app-root>` e impareremo a "decifrare" le righe presenti al suo interno, non appena avremo capito come creare un nuovo componente.

3.3 Come creare il componente Menu dell'app MetroChat

Come ogni sito web che si rispetti, costituito da più pagine, anche nel caso di un'app, non dovremo fermarci al componente predefinito `<app-root>` creato in automatico, ma dovremo imparare a crearne di altri, proprio perché un'app complessa, sarà costituita da diversi di questi, con una tipica struttura ad albero rovesciato.

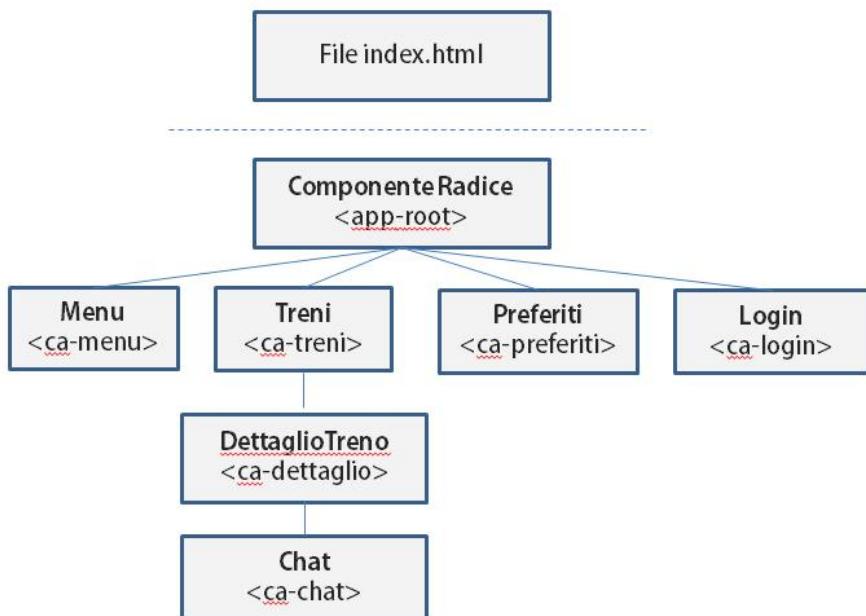


Figura 3.3 – Struttura ad albero dell'applicazione MetroChat, con i vari componenti

In questo modo, con un percorso graduale, impareremo tutti i concetti legati ai componenti, dal nome da dare al selettore, dove inserirlo, come aggiungere un template, come definire la classe e che nome darle.

Il metodo più rapido per creare un nuovo componente, è sfruttare la linea di comando. È chiaro che dovrà prima di tutto creare un nuovo progetto, e poi spostarsi all'interno della cartella `app`. Qui dentro andrai a creare i diversi componenti di cui sarà costituita la tua applicazione, oltre che al file `app.module.ts` già creato in precedenza.

Per creare un nuovo componente, sfruttando la linea di comando, dovrà scrivere:

```
ng g component NOME
```

dove al posto di `NOME` dovrà sostituire l'effettivo nome che vorrai assegnare al componente.

NB: Ricordati di creare i tuoi componenti, all'interno della cartella `app` della tua applicazione

Per la scelta del nome, lo stile consigliato, è di usare la notazione "Upper Camel Case" ossia la prima lettera maiuscola di ogni porzione di parola. Nel caso allora volessimo creare il componente "menu" dell'ipotetica app licazione di fig. 3.1, potremmo scrivere:

```
ng g component Menu
```

Dopo l'esecuzione del comando, apparirà una nuova cartella di nome `menu`, con all'interno una serie di file, compreso quello con nome `menu.component.ts`, che rappresenta proprio il nostro componente. Nota come il nome sia scritto in minuscolo ed è stata aggiunta la parola "component".

Questa è una convenzione cui dovrà abituarti, e che si applicherà anche ad altri elementi di un'app Angular.

Se provassimo ad aprire il file, con sorpresa troveremmo all'interno una serie di righe già create per noi e che inizieremo ad analizzare nelle prossime pagine. All'apparenza potrebbero sembrare complesse, ma fra meno di 20 minuti saprai "decifrarle" senza alcuna difficoltà.

`app/menu/menu.component.ts`

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

@Component({
  selector: 'app-menu',
  templateUrl: './menu.component.html',
  styleUrls: ['./menu.component.css']
})
export class MenuComponent implements OnInit {
  constructor() { }
  ngOnInit() { }
```

```
    }
}
```

Se il progetto è stato creato sfruttando l'opzione *-minimal*, troverai un codice leggermente diverso, ma capiremo le differenze nei prossimi capitoli.

3.4 Tecniche alternative per creare un componente

Una possibile alternativa per la creazione di un componente è copiare un componente già realizzato, ad esempio proprio quello associato al selettore `<app-root>` e incollarlo nella cartella `app` o in una sottocartella.

Chiaramente dovrà poi andare a modificare i nomi interni, ma soprattutto dovrà andare a modificare il file `app.module.ts`, perché è necessario aggiungere all'array abbinato alla proprietà `declarations` del decoratore `@NgModule()`, il nome della classe associata a questo nuovo componente, oltre che importare la classe stessa, con l'istruzione `import`.

Questa operazione è chiamata "**registrazione**" del componente nel Modulo.

Nel 99% dei casi sono sicuro che ti dimenticherai di farla, e questo è il motivo per cui è prefe ribile usare la tecnica precedente, che al contrario, e segue queste operazioni in automatico.

L'ultima alternativa è quella introdotta con l'avvento della versione 15.x di Angular (novembre 2022) che permette di creare componenti "standalone", ossia senza la necessità di dichiararli all'interno del decoratore `@NgModule` nel file centralizzato `app.module.ts`

Un passo in avanti per semplificare la creazione di app licazioni, anche se per molti non c'è un grosso vantaggio. Per questi componenti infatti dovrà specificare le eventuali dipendenze, importandole direttamente nel file. Nel resto del libro useremo la tecnica "classica", ma nulla ti vieta di usare entrambe le forme.

3.5 Definire il nome del componente grazie al "selettore"

Abbiamo detto che per **identificare** il componente, dobbiamo assegnargli un **nome univoco** che potrà essere usato all'interno del codice html della pagina principale o di altri componenti, per richiamare il componente stesso, sotto forma di tag HTML personalizzato (es. `<ca-menu></ca-menu>`)

Abbiamo detto che un componente non è altro che una classe TypeScript, quindi come sempre avremo un costruttore e del codice all'interno. La grossa differenza rispetto alla classica classe TypeScript, è che dovremo dare ulteriori indicazioni ad Angular, per informarlo che si tratta di un componente.

Questo si fa aggiungendo alla classe una particolare riga, che precede il costruttore, e che viene chiamata "Decoratore", un nome che richiama alla mente un'arte antica che ormai sta scomparendo.

Analizzando il codice qui sotto, di cui ho eliminato alcuni pezzi:

```
app/menu/menu.component.ts

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

@Component({
  selector: 'ca-menu', // Qui definisco il nome!
  ...
})

export class MenuComponent {
  constructor() { }
}
```

notiamo - subito prima del costruttore - la presenza del termine `@Component()`.

Ebbene questo termine è specifico del linguaggio TypeScript e assume il nome inglese di "**decorator**" o decoratore, e come detto, è un elemento che permette di assegnare alla classe, specifiche caratteristiche o metadati, un po' come avviene per il tag `<head>`, interno alle pagine html, in cui la presenza dei vari metatag, permette di specificare proprietà della pagina web, come il titolo, la descrizione etc.

Queste caratteristiche sono espresse sotto forma di un oggetto JavaScript, quindi sono racchiuse all'interno delle classiche parentesi graffe.

NB: Affinché Angular sia in grado di capire cosa sia `@Component()` è necessario importare la definizione dalla libreria `@angular/core`. Non preoccuparti più di tanto per ora, perché queste righe sono aggiunte automatico all'atto della creazione del componente con la linea di comando.

La sintassi con cui puoi definire il nome del selettore del componente, da usare all'interno di codice html, è la seguente:

```
<NOMECOMPONENTE></NOMECOMPONENTE>
```

è necessario inserire la prima proprietà selector:

```
@Component({
  // identifico il nome del selettore
  selector: 'ca-menu'
})
```

In questo caso, il nome scelto per il selettore del componente è ca-menu, ma si è libero di chiamarlo come preferisci, rispettando però le regole di stile indicate in precedenza.

NB: Il nome del selettore non deve avere all'interno i simboli < e > dei tag HTML

Fai attenzione alla sintassi: nomeproprietà: 'valore', perché sarà usata anche per altre proprietà.

3.6 Analisi del componente <app-root>

Alla luce di quanto appreso, siamo in grado ora di capire a cosa fa riferimento l'elemento html <app-root> che abbiamo trovato nel file *index.html* non appena è stata creata la prima applicazione Angular.

```
index.html
<!DOCTYPE html>
<head>
  <title>Angular App</title>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1">
</head>
<body>
  <!-- il componente radice, padre di tutti i componenti -->
  <app-root></app-root>
</body>
</html>
```

Abbiamo imparato che questo tag identifica un componente, quindi dovrò emozionarmi a verificare tra la lista di tutti i file che hanno un nome con questa sintassi *nome.component.ts*, se ne esiste uno che fa riferimento a questo tag.

E' facile individuarlo, in quanto per ora la nostra app è costituita da un solo componente, quindi aprendo il file *app.component.ts* scopriremo dove è stato usato:

```
app/app.component.ts

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

@Component({
  // 1) qui è stato definito il nome del componente
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})

export class AppComponent {
  title = 'app!';
}
```

NB: Il nome impostato per il selettore del componente, tipicamente non deve coincidere con il nome della classe, anche se conviene sceglierlo in modo che sia facile da ricordare, in funzione dell'uso che se ne farà all'interno dell'applicazione.

NB: Il nome del selettore, devi includerlo all'interno di singole virgolette.

La domanda che potresti farti adesso è: come mai, all'apertura dell'app nel browser, è visualizzata la scritta “Welcome to app!” se nel file *index.html* non è presente?

Questo lo capiremo con l'introduzione del concetto di “Template” o “Vista”.

3.7 Definire la rappresentazione grafica: Template

Vediamo di rispondere alla domanda precedente, partendo dal componente con selettore `<ca-menu>` creato in precedenza, che ci permetterà di entrare nel vivo dello sviluppo della nostra applicazione.

La domanda che devi farti è: come posso rappresentare graficamente il componente rappresentativo del menu della mia applicazione? Beh, se stessi creando una classica pagina HTML, partirei con definire un foglio di stile associato e del codice HTML.

Preoccupandoci per ora solo del codice html, potremmo scrivere:

```
<!-- menu applicazione -->
<nav>
  <ul>
    <li><a href="">Link Menu 1</a></li>
```

(continua)

```
<li><a href="">Link Menu 2</a></li>
<li><a href="">Link Menu 3</a></li>
</ul>
</nav>
```

Ora sapendo che con il decoratore `@Component()` posso aggiungere una serie di proprietà alla classe (*metadati*), i progettisti di Angular hanno predisposto un'ulteriore proprietà dal nome `template`, o in alternativa `templateUrl`, finalizzata all'inclusione di codice HTML.

La differenza tra le due è che la prima, ci permette di aggiungere il codice HTML, direttamente nel file della classe del componente, la seconda invece, sfruttando il riferimento a un file con estensione `.html` che andremo a inserire nel progetto.

Sempre nell'ipotesi del componente `<ca-menu>`, nel primo caso scriveremo:

app/menu/menu.component.ts

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

@Component({
  // 1) Definisco il nome del selettore
  selector: 'ca-menu',
  // 2) Definisco il template grafico "inline"
  template: `
    <!-- menu applicazione -->
    <nav>
      <ul>
        <li><a href="">Link Menu 1</a></li>
        <li><a href="">Link Menu 1</a></li>
        <li><a href="">Link Menu 1</a></li>
      </ul>
    </nav>
  `
})

export class MenuComponent {
  constructor() { }
}
```

NB: Osserva come la stringa rappresentativa del codice html della pagina, sia stata inserita tra i due simboli detti "backtick" che sono diversi dal classico simbolo di singola virgoletta. Per poterli inserire usando la tastiera italiana, dovrà digitare la combinazione di tasti: ALT e contemporaneamente digitare il numero 96 (ALT+96)

Sfruttando la proprietà `template`, sicuramente si ha il vantaggio di poter visualizzare il codice HTML del componente e di poter apportare modifiche alla logica che inserirai nel corpo della classe, senza dovere aprire un altro file.

Per applicazioni più complesse, che hanno molti elementi da rappresentare per singolo componente, di norma si lavora con un file esterno, dove si andrà ad inserire il codice HTML rappresentativo del componente, nel nostro caso del menu.

Dovrai però sfruttare la proprietà `templateUrl`, come visualizzato qui sotto:

```
app/menu/menu.component.ts

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

@Component({
  // 1) Definisco il nome del selettore
  selector: 'ca-menu',
  // 2) Definisco il template grafico
  templateUrl: './menu.component.html'
})

export class MenuComponent {
  constructor() { }
}
```

dove `menu.component.html`, conterrà proprio il codice per rappresentare graficamente nel browser il menu, come indicato qui sotto:

```
app/menu/menu.component.html

<!-- menu applicazione -->
<nav>
  <ul>
    <li><a href="">Link Menu 1</a></li>
    <li><a href="">Link Menu 2</a></li>
    <li><a href="">Link Menu 3</a></li>
  </ul>
</nav>
```

NB: Il percorso a cui fa riferimento il file `menu.component.html`, è relativo alla posizione dello stessa classe. E' prassi infatti tenere i file rappresentativi di un componente, raggruppati all'interno della stessa cartella.

Pertanto quello che farà Angular, non appena incontrerà un selettore collegato ad un componente, sarà di sostituirgli il codice HTML definito nella proprietà `template` o `templateUrl`.

3.8 Come aggiungere un foglio di stile

Altra proprietà che puoi sfruttare per aggiungere delle regole CSS da applicare solo alla sezione rappresentata dal componente, è costituita da `styleUrls`. Anche in questo caso dovrà definire un file esterno, che ospiterà tutte le regole CSS da usare per il componente.

Questa proprietà in realtà puoi anche ometterla per i componenti interni, perché spesso si userà un foglio di stile centralizzato, per evitare di dover spezzettare le regole in decine di file difficili da tenere aggiornati. Questo nell'ipotesi il foglio di stile non sia particolarmente "pesante" dal punto di vista dei kB da scaricare.

3.9 Il template del componente <app-root>

Ora che sappiamo come si definisce il template del componente, ossia la propria rappresentazione a livello di codice HTML, vediamo come sia stata progettata la grafica del componente principale dell'applicazione.

Se proviamo ad aprire il file `app.component.ts`, troveremo la proprietà `templateUrl` valorizzata con `'./app.component.html'`.

```
app/app.component.ts

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

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

export class AppComponent {
  title = 'app!';
}
```

All'interno di quest'ultimo file, troveremo, tra le diverse righe di codice HTML, il seguente pezzo:

```
<span>{{ title }} app is running!</span>
```

E' chiaro allora il motivo per cui, non appena hai aperto nel browser la tua prima app Angular, compariva una scritta con uno sfondo simile a quella della figura seguente.

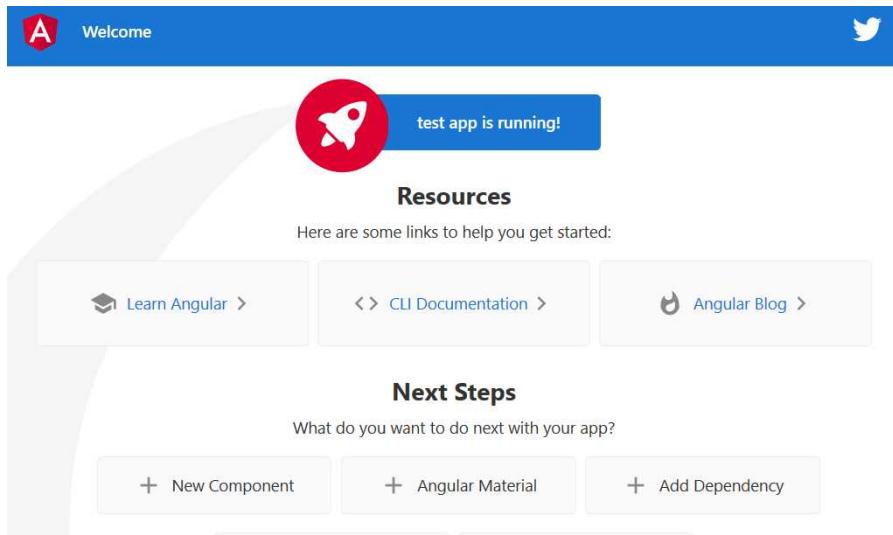


Figura 3.4 – Schermata iniziale applicazione Angular, visualizzate nel browser

Rimane da capire da dove viene prelevata la stringa "MiaApp ..."! e capire la strana notazione presente all'interno del codice html, costituita dai simboli {{}}, ma penso tu abbia già intuito qualcosa.

Ora che l'applicazione ha ben due componenti, potremmo inserire il componente con selettore <ca-menu> internamente al template del componente radice, in modo da sperimentare la costruzione di un "**albero di componenti**", caratteristico di ogni app Angular.

Le modifiche a livello di codice html, potrebbero essere di questo tipo:

```
<ca-menu><ca-menu>
<h1> Welcome to {{title}} </h1>
```

mentre a livello di classe del componente nulla cambierà rispetto a prima:

```
app/app.component.ts
import { Component } from '@angular/core';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  (continua)
```

```

    styleUrls: ['./app.component.css']
  })

export class AppComponent {
  title = 'app!!';
}

```

Quello che ottengo aprendo l'applicazione nel browser, sarà proprio la lista dei menu appena creata, con sotto la scritta “Welcome to app!!”.

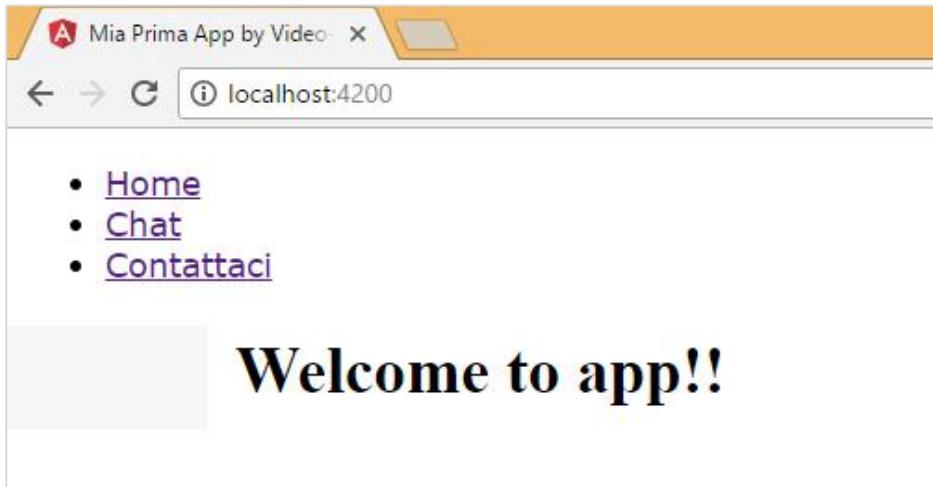


Figura 3.5 – Schermata componente AppComponent personalizzato

Siamo riusciti a incastrare il nostro primo componente `MenuComponent`, all'interno di un altro componente, in particolare all'interno del componente padre con selettore `<app-root>`

A questo punto la domanda che potresti farti è: come fa Angular a rappresentare il componente *menu*, se nella classe del componente principale, non vi è alcuna indicazione di questo tipo?

Ebbene, se ti ricordi, ogni qualvolta si crea un nuovo componente, questo viene aggiunto all'interno del file `app.module.ts`, in particolare all'interno dell'array della proprietà `declarations` del decoratore `@NgModule()`.

Quest'ultimo discorso vale solo se il componente non è stato definito come “standalone”, ossia con le modalità introdotte a partire dalla versione 15 di Angular, che come dicevamo in precedenza, ci permettono di bypassare `NgModule`.

Ebbene con i componenti “classici”, se provassimo ad eliminarlo dal file app.module.ts, come indicato qui sotto:

```
app/app.module.ts

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

import { MiaAppComponent } from './app.component';
//import { MenuComponent } from './menu/menu.component';

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

sarebbe generato un errore, proprio perché Angular non sarebbe in grado di capire come rappresentare il componente <ca-menu>.

3.10 Definire la logica del componente

Arrivati a questo punto, l'ultima cosa che ci rimane da vedere è come definire il **cuore** del componente, ossia le sue **funzionalità**.

Un componente, nella sua forma più banale, potrebbe solo mostrare del codice html, corrispondente a quello inserito nel template, ma nelle maggior parte delle applicazioni, dovrà anche interagire con le azioni fatte dall'utente o mostrare dei dati provenienti da un database remoto.

Ecco allora che devo poter capire come creare una relazione tra gli elementi definiti nel template ed eventuali azioni da far fare.

Queste funzionalità dovranno essere progettate sfruttando proprio TypeScript come linguaggio, quindi per grossa parte del codice il classico JavaScript.

Sapendo che ogni componente è una classe, la logica dovrà essere inserita all'interno del corpo della classe, che chiaramente dovrà avere un nome, corrispondente proprio a quello scelto in fase di creazione del componente con la linea di comando, (es. *MenuComponent*)

Il risultato finale di tutto il nostro sforzo per definire i tre elementi principali del componente con selettore <ca-menu>, sarà rappresentato qui sotto:

```
app/menu/menu.component.ts

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

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

export class MenuComponent {
  // qui dentro definisco la logica del componente
  // ed eventuali operazioni di inizializzazione dati
  constructor()
}
```

Nota la presenza della parola chiave `export` che precede la parola `class`. Questa serve nel caso abbia la necessità di importare la classe anche in altre sezioni della mia applicazione.

Il passo successivo sarà quello di iniziare a sperimentare come si possa creare una "comunicazione" tra le righe presenti all'interno della classe e il template, perch è spesso avrò la necessità di mostrare dei dati provenienti da sorgenti esterne, operazioni che dovrò chiaramente fare all'interno della classe.

3.11 Popolare il template con proprietà della classe

Ora che siamo arrivati al cuore del componente , iniziamo a divertirci un po', vedendo cosa sia possibile fare.

Partiamo con un esempio banale, per cercare di capire come avviene il meccanismo di comunicazione tra template e corpo della classe detto anche "**Data Binding**".

Se ad esempio, volessi creare un menu, con il testo dei link programmato direttamente nel codice della classe, come potresti fare?

Beh, noi sappiamo che in JavaScript, si possono definire delle variabili, quindi il primo passo potrebbe essere quello di definire tre **variabili interne alla classe**.

Nella terminologia della programmazione ad oggetti, queste ultime vengono chiamate anche proprietà o membri.

app/menu/menu.component.ts

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

@Component({
  selector: 'ca-menu',
  templateUrl: './menu.component.html',
  styleUrls: ['./menu.component.css']
})
export class MenuComponent {
  // definisco 3 proprietà del componente
  link_menu_1;
  link_menu_2;
  link_menu_3;
  constructor() {
    this.link_menu_1 = 'Treni';
    this.link_menu_2 = 'Preferiti';
    this.link_menu_3 = 'Login';
  }
}
```

TypeScript ci permette di associare ad ogni variabile anche la tipologia di dato che andrà a "contenere", nel nostro caso una stringa. Pertanto abituiamoci fin da ora a specificarlo con la notazione `nomevariabile: TIPO`, per le tre proprietà `link_menu_N` scelte:

app/menu/menu.component.ts

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

@Component({
  selector: 'ca-menu',
  templateUrl: './menu.component.html',
  styleUrls: ['./menu.component.css']
})
export class MenuComponent {
  // definisco 3 proprietà del componente
  link_menu_1:string;
  link_menu_2:string;
  link_menu_3:string;
  constructor() {
    this.link_menu_1 = 'Treni';
    this.link_menu_2 = 'Preferiti';
    this.link_menu_3 = 'Login';
  }
}
```

E' buona norma abituarsi a **inizializzare** le proprietà direttamente nel costruttore, perché è il primo pezzo di codice eseguito all'atto del caricamento del componente (vedi par. 3.14 sul ciclo di vita di un'app)

In assenza di tale passaggio, e nell'ipotesi di ripristinare le configurazioni di compilazioni predefinite modificate nel Capitolo 1, sarebbe segnalato il classico errore TypeScript di inizializzazione:

```
Error: src/app/app.component.ts:10:3 - error TS2564: Property 'link_menu_1' has no initializer and is not definitely assigned in the constructor.
10   link_menu_1:string;
```

Non spaventarti quindi se durante i test dovessi vedere questo messaggio, perché per rendere il codice meno prolioso, in alcuni punti ho volutamente saltato tutta la fase di inizializzazione delle proprietà dei componenti.

3.12 Definire i "segnaposto" nel template con l'interpolazione

Come fare ora per visualizzare il contenuto di queste tre variabili all'interno del template? Se provassi a inserire il solo nome di queste, verrebbe visualizzata una stringa corrispondente al nome della variabile e non al suo contenuto.

E' necessario allora usare la tecnica chiamata **"interpolazione"**, una delle tante tecniche che vedremo per manipolare il contenuto del template.

In sostanza si tratta di inserire nel template, il nome della variabile definita nella classe, "contornata" da due parentesi graffe in apertura e chiusura: una sorta di "segnaposto":

```
{ {NOMEVARIABILE} }
```

Ogni qualvolta Angular trova questa notazione nel template, si ricorda che dovrà andare a sostituire il corrispondente valore prelevato dalla rispettiva variabile definita nel corpo della classe del componente.

Solo le variabili di quel componente possono essere visualizzate nel relativo template. Non possono visualizzare variabili definite in altri componenti figli o padri, a meno che non applichi ulteriori tecniche che vedremo quando parleremo di progettazione di componenti intelligenti.

Il template del componente menu *menu.component.html*, diventerebbe così:

app/menu/menu.component.html

```
<!-- menu applicazione -->
<nav>
  <ul>
    <li><a href="">{{link_menu_1}}</a></li>
    <li><a href="">{{link_menu_2}}</a></li>
    <li><a href="">{{link_menu_3}}</a></li>
  </ul>
</nav>
```

Le due parentesi di apertura e chiusura, devono essere inserite SENZA spazi tra le due parentesi, mentre il nome della variabile interna può essere inserito con spazi

Qui sotto alcuni esempi di cosa è possibile e non è possibile fare:

app/menu/menu.component.html

```
<!-- menu applicazione -->
<nav>
  <ul>
    <li><a href="">{ link_menu_1 }</a></li>      <!-- NO -->
    <li><a href="">{{ link_menu_2 }}</a></li>      <!-- Si -->
    <li><a href=""> {{ link_menu_3 }} </a></li>  <!-- Si -->
  </ul>
</nav>
```

Il risultato finale , non appena il componente è visualizzato nel browser, sarà che al posto dei tre "segnaposto", saranno sostituiti i valori inseriti nelle tre variabili, ossia rispettivamente le stringhe “Treni”, “Preferiti” e “Login”.

3.13 Inserire espressioni all'interno del "segnaposto"

All'interno del segnaposto evidenziato dalle doppie parentesi graffe, non ci si limita a inserire variabili di tipo *stringa*, ma è possibile inserire un elemento di un *Array*, oppure un'espressione, come nell'esempio qui sotto.

```
<button>Pulsante LUCE : {{ isActive ? "ON" : "OFF" }}</button>
```

All'interno delle parentesi graffe, ho inserito l' a classica istruzione `if then else` (operatore ternario), per aggiungere alla stringa di testo "Pulsante LUCE:" il valore "ON" o "OFF" sulla base del valore della variabile `isActive`, membro della classe.

Angular pertanto è in grado di valutare un'espressi one semplice inserita interna alle doppie parentesi graffe. Per capire come collegare un evento alla pressione sul bottone, dovremo aspettare il tutorial sugli gestione degli eventi.

Avrai notato che, nel codice del template precedente, sono ripetuti tre tag ``. Per un menu, di cui sappiamo il numero massimo di voci da visualizzare, è una situazione abbastanza normale, ma quando abbiamo la necessità di visualizzare dei dati provenienti da fonti esterne, quali database o file JSON, la situazione cambia.

Infatti, non conosceremo a priori il numero di dati restituiti, e come conseguenza, non riusciremo a impostare un layout con N tag ripetuti , se il valore di N appunto non è noto.

Per questo motivo dobbiamo conoscere altre tecniche in grado di aiutarci nella manipolazione del DOM.

3.14 Ciclo di vita di un'app Angular

Il caricamento di un'app Angular, presuppone il caricamento di N componenti, ognuno dei quali ha precisi "stati" durante il **ciclo di vita dell'intera app**.

Per capire il concetto di "stato", possiamo ricorrere all'analogia delle fermate di un autobus, da un capolinea all'altro. Durante il percorso, è possibile salire sull'autobus, in corrispondenza alle diverse fermate.

Nel caso di un'app Angular, queste "fermate" sono gestite in automatico e ci permettono di entrare o agganciarci a diversi stati d'esecuzione di un'app.

Sappiamo infatti, dai capitoli precedenti, che il cuore di ogni componente è costituito dal corpo della classe. E all'interno di ogni classe TypeScript, ci sarà sempre il costruttore, dove per convenzione ed efficienza, si inizializzano le eventuali proprietà del componente.

Il costruttore, quindi, è proprio il primo elemento che viene richiamato all'atto del caricamento di un componente.

Questo è il motivo per cui, nel caso in cui il componente dipenda da altri elementi, si sfrutta il costruttore per "iniettare" le dipendenze.

Ad esempio, quando un componente o service ha la necessità di comunicare con la rete, dovremo iniettare nel costruttore un service di nome http, grazie al quale potremo accedere ad una serie di metodi della libreria `HttpClient`, utili per dialogare con la rete.

Ecco allora la lista degli eventi principali, che nel corso dello sviluppo di un'app, potrai sfruttare per effettuare degli interventi sui dati oper agire sulla logica di funzionamento dell'applicazione.

E' chiaro che per poterli utilizzare, dovrai implementarli nel costruttore del componente, e includerli richiamandoli dalla libreria `@angular/core`.

ngOnInit

Il metodo `ngOnInit` è chiamato sempre **una sola volta dopo l'esecuzione del costruttore** e subito dopo la prima chiamata al metodo `ngOnChanges`. Può essere sfruttato per inizializzare alcune proprietà interne al componente o richiamare metodi di un *service*, per operazioni di scambio dati con la rete. All'interno del corpo della classe del componente, dovrai usare la notazione:

```
class MioComponente implements OnInit {
  constructor() {}
  ngOnInit() {
    // Definisco qui le azioni da fare
  }
}
```

ngOnChanges

Quando un componente riceve dei dati di ingresso che cambiano nel tempo e sulla base dei quali devi eseguire determinate azioni, ci viene in aiuto il metodo `ngOnChanges`, per segnalare tutte le volte in cui il valore subisce una modifica.

Il metodo riceve un parametro in ingresso, che è un oggetto di tipo `SimpleChange`, il quale descrive nel dettaglio le variazioni di ogni proprietà di input, grazie a tre proprietà (`isFirstChange`, `previousValue`, `currentValue`).

Un possibile oggetto collegato alla proprietà di ingresso "nome", potrebbe essere:

```
{"nome": {"previousValue": "", "currentValue": "Davide"} }
```

La proprietà d'ingresso cambia dalla stringa vuota "", al valore " Davide". Per intercettare tale cambiamento, potrei scrivere:

```
class MioComponente implements OnChanges {
  constructor() {}
  ngOnChanges(changes: SimpleChange) {
    if(changes['nome'].isFirstChange()) {
      console.log('Primo cambio di valore');
    }
    // Mostro l'attuale valore e quello precedente di nome
    console.log('Valore attuale: ' + changes['nome'].currentValue);
    console.log('Valore prec. ' + changes['nome'].previousValue);
  }
}
```

ngDoCheck

`ngDoCheck` viene attivato ogni volta che le proprietà d'ingresso di un componente o una direttiva vengono controllati. È possibile utilizzare questo aggancio al ciclo di vita per estendere il controllo con la tua logica personalizzata.

ngAfterContentInit

Chiamato dopo `ngOnInit` quando il contenuto di un componente o una direttiva è stato inizializzato.

ngAfterContentChecked

Chiamato dopo ogni verifica del contenuto del componente o di una direttiva.

ngAfterViewInit

Chiamato dopo `ngAfterContentInit` quando la vista del componente è stata inizializzata. Si applica solo a componenti.

ngAfterViewChecked

Chiamato dopo ogni verifica della vista di un componente. Si applica solo ai componenti.

ngOnDestroy - L'ultimo metodo che viene chiamato nel ciclo di vita di un componente, appena prima che l'istanza del componente sia finalmente distrutta.

Capitolo 4

Manipolare il DOM con le direttive

4.1 Ripetere elementi del DOM: la direttiva *ngFor

Angular offre una serie di strumenti per manipolare il DOM e potenziare i tag dell'HTML, di funzionalità non presenti nel linguaggio nativo. A questo scopo, in Angular, sono state introdotte le “**Direttive**”. Non ti devi spaventare dal nome, perché si tratta semplicemente di un insieme di notazioni, da incastrare all'interno di un normale tag HTML e che permettono di arricchirlo di nuove funzionalità.

Le direttive si distinguono tra quelle in grado di cambiare la struttura del DOM della pagina (direttive “strutturali”), e quelle che in grado di cambiare solo l'aspetto di tag già presenti (direttive “attributo”), come ad esempio il colore di un testo.

Non appena si utilizzano le direttive, si inizia a creare quel collegamento tra il **template** e il **corpo della classe**, che si manifesterà in tutta la sua potenza non appena parleremo dei form.

Si parla in gergo tecnico di “**Template binding**” intendendo il collegamento di proprietà di una classe, con proprietà presenti nel template del componente.

Non devi confondere la proprietà di un elemento del DOM con gli attributi di un tag HTML, che sono quelli con cui generalmente siamo abituati a lavorare quando si sviluppano le classiche pagine web. E' una sottigliezza che andrebbe capita a fondo, ma per non deviare troppo dal nostro obiettivo, memorizza questo concetto:

“Il Template binding lavora con le proprietà di un elemento del DOM, di un componente, o direttiva e NON con gli attributi.”

Tornando a noi, una delle direttive più usate in Angular, è quella che ti permetterà di modificare il DOM ripetendo un certo numero di volte uno specifico tag HTML o componente. Sto parlando della direttiva `*ngFor`.

E' frequente infatti creare delle applicazioni che visualizzino un insieme ripetuto di articoli, dotati di immagine, titolo, descrizione. L'insieme di questi articoli, potrebbe essere inserito all'interno di un tag `<article>`, oppure all'interno di un semplice `<div>`, come nella rappresentazione qui sotto:

```
<div class="news">
  <ul>
    <li><h2>Titolo Notizia 1 </h2><p>Desc. notizia 1</p></li>
    <li><h2>Titolo Notizia 2 </h2><p>Desc. notizia 2</p></li>
    <li><h2>Titolo Notizia 3 </h2><p>Desc. notizia 3</p></li>
    <li><h2>Titolo Notizia 4 </h2><p>Desc. notizia 4</p></li>
  </ul>
</div>
```

che sarà visualizzato nel browser in questo modo:

- **Titolo Notizia 1**
- **Titolo Notizia 2**
- **Titolo Notizia 3**
- **Titolo Notizia 4**

Figura 4.1 – Visualizzazione elenco notizie nel browser

In un'applicazione reale, non saprai a priori quanti e quali articoli dovrai aggiungere, perché questi saranno tipicamente prelevati da una sorgente esterna all'applicazione (es. database o servizio esterno).

Pertanto non potrai inserire i vari segnaposto, come visto nel tutorial precedente.

Entra in azione allora la direttiva `*ngFor`.

Nell'ipotesi di aver progettato il componente con nome `<ca-listanews>`, nel seguente modo:

notizie/notizie.component.ts

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

@Component({
  selector: 'ca-listanews',
  template: `
    <div class="news">
      <ul>
        <li><h2>Titolo Notizia 1 </h2></li>
        <li><h2>Titolo Notizia 2 </h2></li>
        <li><h2>Titolo Notizia 3 </h2></li>
      </ul>
    </div>`})
export class NotizieComponent { }
```

la domanda che potresti farti è: come posso iniettare delle notizie nel template, senza conoscere a priori il numero?

4.2 Sintassi della direttiva *ngFor

Grazie a questa direttiva, io riesco a ripetere determinati elementi del DOM un certo numero di volte. La sintassi base è la seguente:

```
<nometag *ngFor="let variabile of listadati">
  ...
</nometag>
```

Il simbolo * deve essere inserito attaccato a `ngFor`. Fai poi attenzione a rispettare le minuscole e maiuscole. Questa è la sintassi base, ma altre caratteristiche le puoi consultare collegandoti alla pagina: <https://angular.io/api/common/NgForOf>

A destra dell'uguale, all'interno delle doppie virgolette, si inserisce un'espressione che stabilirà il numero totale di ripetizioni:

```
let X of Y
```

X è il nome di una variabile locale da usare nel template, mentre Y è un array, definito nel corpo della classe, che conterrà l'insieme dei valori con cui valorizzare la variabile X ad ogni ripetizione. Se hai già esperienza con altri linguaggi di

programmazione, la notazione ricorda quella che si usa per i classici “cicli for”, da cui è stato copiato il nome.

Spesso avrai a che fare con array di stringhe o numeri ma avremo modo di vedere anche come manipolare array di oggetti.

Quello che viene ripetuto nel DOM, è il tag a cui è abbinata la direttiva (comprensivo di eventuali figli), per un numero di volte pari al numero di elementi presenti nell'array.

4.3 Uso della direttiva `*ngFor` con un array

Ipotizzando di aver popolato un array, di nome `listanews`, con notizie prelevate in qualche modo da una sorgente esterna e ipotizzando per ora di inserirle manualmente nell'array in questo modo:

```
listanews= ['Titolo Notizia 1','Titolo Notizia 2', 'Titolo  
Notizia 3', 'Titolo Notizia 4'];
```

allora per visualizzare tutte le notizie, sempre nell'ipotesi che non conosca il numero totale presente in `listanews`, potrei sfruttare la direttiva `*ngFor` applicata a ``:

notizie/notizie.component.ts

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

@Component({
  selector: 'ca-listanews',
  template: `
    <div class="news">
      <ul>
        // elemento che si ripete N volte
        <li *ngFor="let titolo of listanews">
          <h2>{{titolo}}</h2>
        </li>
      </ul>
    </div>`})
export class NotizieComponent {
  listanews = ['Titolo Notizia 1','Titolo Notizia 2',  
 'Titolo Notizia 3', 'Titolo Notizia 4'];
}
```

A seconda di quante notizie saranno memorizzate all'interno dell'array `listanews`, la direttiva andrà a modificare il DOM creando tanti tag `` con all'intero un segnaposto `{{titolo}}`, valorizzato proprio con il valore del relativo elemento dell'array che si sta percorrendo.

Non appena aggiungi un nuovo elemento all'array `listanews`, in automatico si aggiornerà anche il template, senza che tu debba intervenire manualmente sul codice html.

Se questo componente lo aggiungessi al template del componente radice dell'applicazione, `app.component.ts`:

```
<ca-menu></ca-menu>
<h1>{{title}}</h1>
<ca-listanews></ca-listanews>
```

otterresti la visualizzazione di tutte le news sotto il menu , come evidenziato qui sotto:

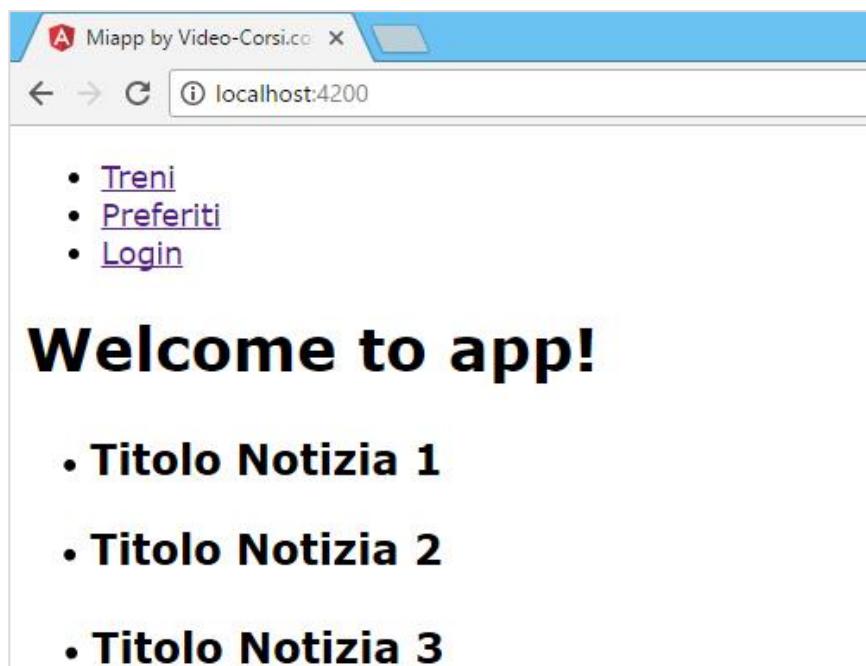


Figura 4.2 – Visualizzazione notizie interne al componente NotizieComponent

4.4 Uso della direttiva *ngFor con un oggetto JSON

Se al posto dell'array avessi un oggetto JSON cosa cambierebbe? Non ci sono particolari avvertenze. Si tratta solo di fare attenzione a come è strutturato l'oggetto JSON che contiene i dati da scorrere.

Spesso infatti i dati recuperati da sorgenti esterne, sono proprio in formato JSON, quindi avere già un'infarinatura di come si dovranno gestire, ci può essere d'aiuto per capire al meglio come sfruttare la potenza di Angular.

Partiamo come sempre dalla rappresentazione dei dati, e poi vediamo come creare il componente con il relativo template.

```
listanews = [{  
    titolo: 'Titolo Notizia 1',  
    descrizione: 'Descrizione Notizia 1'  
},  
{  
    titolo: 'Titolo Notizia 2',  
    descrizione: 'Descrizione Notizia 2'  
},  
{  
    titolo: 'Titolo Notizia 3',  
    descrizione: 'Descrizione Notizia 3'  
},  
{  
    titolo: 'Titolo Notizia 4',  
    descrizione: 'Descrizione Notizia 4'  
}];
```

Come puoi osservare si tratta della rappresentazione di un array di oggetti JavaScript, con chiavi `titolo` e `descrizione`.

La direttiva `*ngFor` in questo caso, scorrerà sempre tra i diversi elementi dell'array, solo che ora la variabile locale al template, sarà valorizzata con un oggetto JavaScript.

Trattandosi di un oggetto, dovrò utilizzare la tipica **notazione del punto**, per accedere alle diverse chiavi o proprietà dell'oggetto.

notizie/notizie.component.ts

```

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

@Component({
    selector: 'ca-listanews',
    template: `
        <div class="news">
        <ul>
            <li *ngFor="let notizia of listanews">
                // notizia è un oggetto JS
                <h2>{{notizia.titolo}}</h2>
                <p>{{notizia.descrizione}}</p>
            </li>
        </ul>
    </div> `
})
export class NotizieComponent {
    listanews = [
        {
            titolo: 'Titolo Notizia 1',
            descrizione: 'Descrizione Notizia 1'
        },
        {
            titolo: 'Titolo Notizia 2',
            descrizione: 'Descrizione Notizia 2'
        },
        {
            titolo: 'Titolo Notizia 3',
            descrizione: 'Descrizione Notizia 3'
        },
        {
            titolo: 'Titolo Notizia 4',
            descrizione: 'Descrizione Notizia 4'
        }
    ];
}

```

Abbiamo pertanto raggiunto lo stesso risultato visto in precedenza, con l'unica differenza che nel template è stato aggiunto anche un nuovo paragrafo con all'interno la visualizzazione della chiave `descrizione`.

Insomma i primi passi per poter visualizzare dei dati recuperati in modo asincrono da una sorgente esterna via HTTP.

Negli esempi visti, non abbiamo sfruttato la peculiarità di TypeScript di definire la tipologia di dati per ogni variabile.

Potremmo definire `listanews` con tipo `<any>`, anche se avrebbe più senso andare a definire una tipologia di dati specifica che potremmo chiamare `News`. Cosa che impareremo a fare in uno dei prossimi capitoli.

4.5 Creiamo il componente **Treni** dell'app MetroChat

Vediamo di applicare questi concetti anche all'applicazione che stiamo progettando. La schermata iniziale che l'utente visualizza non appena accede all'applicazione, nell'ipotesi abbia già eseguito l'accesso con i propri dati, è una lista di treni in arrivo nell'ipotetica stazione geolocalizzata in funzione della sua attuale posizione.

Ipotizzando per ora d'inserire manualmente questi dati, così come fatto per il componente **Notizie**, potresti creare un array di nome `listametro` e sfruttare le nozioni appena apprese, per mostrarli in sequenza.

```
listametro = [
    {idt:'ASD', linea:'Rossa', numchatting:32, tempo:125000},
    {idt:'AKE', linea:'Verde', numchatting:29, tempo:145000},
    {idt:'BFD', linea:'Gialla', numchatting:47, tempo:155000}
];
```

Il componente lo devi creare all'interno della cartella principale del progetto sfruttando la linea di comando:

```
ng g component treni
```

che aggiungerà una cartella **treni** e il file *treni.component.ts*.

Ti ricordo che in automatico Angular modifica anche il file *app.module*, aggiungendo il nuovo componente all'array associato alla proprietà `declarations` di `@NgModule()`, e in più la relativa riga di `import`.

Modificando il nome del selettore in `<ca-treni>` ottengo:

```
treni/treni.component.ts

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

@Component({
  selector: 'ca-treni',
  template: `
    <div class="listatreni">
      <ul>
        <li *ngFor="let metro of listametro">
          <h2>{{metro.linea}}</h2>
          <p>ID: {{metro.idt}}</p>
          <p>Partenza tra: {{metro.tempo}}</p>
        </li>
      </ul>
    </div>
  `
```

(continua)

```

        </div>
    }

export class TreniComponent {
  listametro = [
    {idt:'ASD', linea:'Rossa', numchatting:32, tempo:125000},
    {idt:'AKE', linea:'Verde', numchatting:29, tempo:145000},
    {idt:'BFD', linea:'Gialla', numchatting:47, tempo:155000}
  ];
}

```

Come puoi vedere, il componente mostra i dati come il nome della linea, l'identificativo associato al treno e il tempo rimanente alla partenza, che vedremo sarà espresso in millisecondi.

Per testare il funzionamento del componente, dobbiamo aggiungerlo al template del componente principale dell'app, ossia *app.component.ts*, in quanto ancora non abbiamo visto come si naviga tra le diverse sezioni di un'app Angular.

Potremmo inserirlo come se fosse un normale tag html dal nome `<ca-treni>`, subito sotto il componente `Menu` inserito in precedenza:

```

<ca-menu></ca-menu>
<h1>MetroChat {{title}}</h1>
<ca-treni></ca-treni>

```

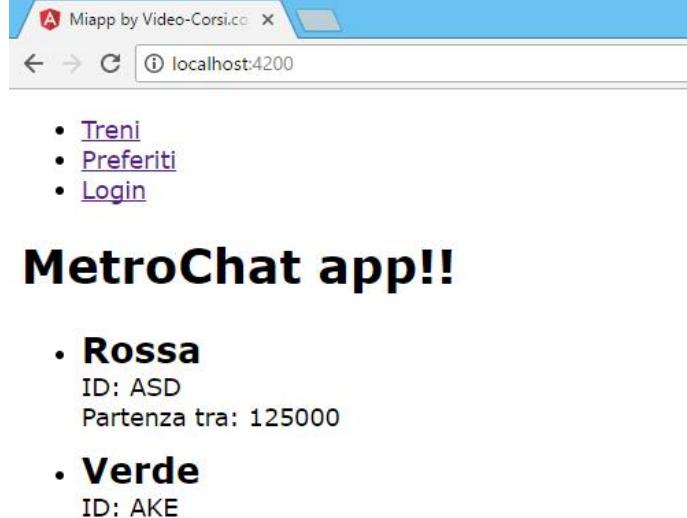


Figura 4.3 – Visualizzazione componente TreniComponent

4.6 Visualizzare o nascondere elementi del DOM: direttiva *ngIf

Altra direttiva importante è quella che ci servirà per indicare se una determinata stanza è stata selezionata come preferita o meno, quindi in generale per mostrare o meno un elemento del DOM. Vediamo un esempio legato alla domotica per rendere le cose più "luminose".

Ipotizziamo tu voglia progettare un'interfaccia in grado di visualizzare all'utente l'elenco di tutte le stanze della casa con abbinata un'icona a forma di lampadina nello stato acceso o spento.

Una prima bozza di pagina, potrebbe essere simile a quella qui sotto, in cui elenco all'interno di un tag ``, i nomi delle diverse stanze, abbinando l'icona di una lampadina, solo per le stanze attualmente illuminate (fig. 4.4):

```
<div class="lucistanze">
<ul>
  <li>Luce Cucine</li>
  <li><i class="material-icons on">lightbulb_outline</i> Luce Pranzo
  </li>
  <li>Luce Bagno</li>
</ul>
</div>
```

dove, grazie all'ausilio dell'attributo di classe "on", sono in grado colorare lo sfondo della lampadina e indicare così all'utente che la particolare stanza è illuminata.

L'icona della lampadina è stata inserita con un tag ``, sfruttando il "font material" di Google, quindi inserendo la seguente riga, nei meta tag del file `index.html`:

```
<link href="https://fonts.googleapis.com/icon?family=Material+Icons"
      rel="stylesheet">
```

Il risultato è simile a questo:

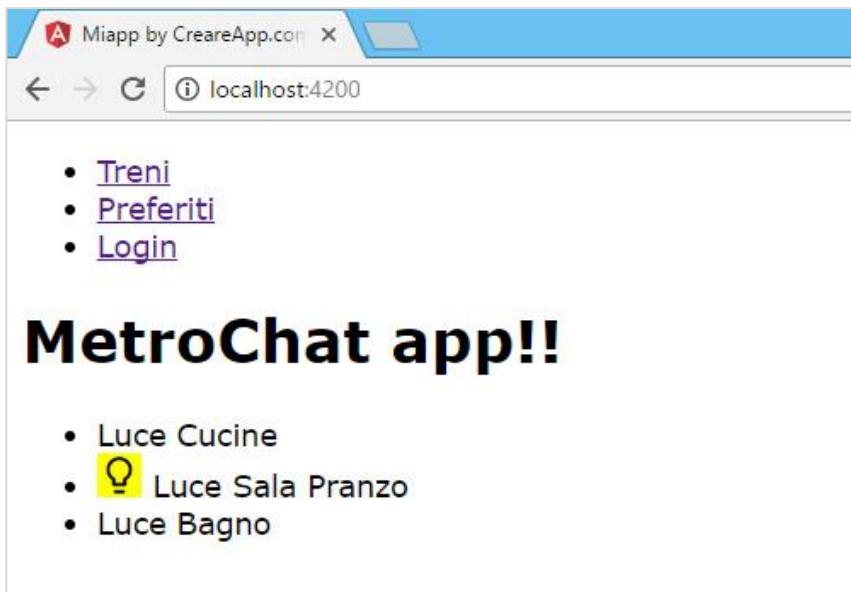


Figura 4.4 – Inserimento di un tag <i> per visualizzare una lampadina accesa/spenta

L'unica differenza tra una riga e l'altra, è data dalla presenza o meno del tag <i> a cui è associata la visualizzazione della lampadina.

Ebbene in Angular è possibile usare una direttiva che ci permette di aggiungere alcuni elementi nel DOM, sulla base di una condizione. Si tratta della **direttiva *ngIf**, che ha una sintassi simile a questa:

```
<taghtml *ngIf="condizione o espressione">...</taghtml>
```

Osserva la presenza del simbolo * e come la lettera i sia stata scritta in maiuscolo, il tutto senza inserire spazi tra l'asterisco e la prima lettera.

Nell'ipotesi la condizione inserita all'interno delle doppie virgolette sia vera, il tag html a cui è applicata la direttiva, sarà visualizzato nella pagina.

Questo è uno dei possibili comportamenti ma a seconda della condizione, io posso visualizzare un blocco con associata una specifica "etichetta" o in alternativa, un altro blocco, seguendo il classico stile dell'istruzione if, else di molti linguaggi di programmazione.

4.7 Esempio d'uso della direttiva `*ngIf`

Abbiamo già imparato a ripetere ciclicamente un particolare elemento del DOM, sulla base di valori memorizzati all'interno di un array.

Visto che nell'esempio analizzato in precedenza il tag `` si ripete più volte, potrei sfruttare la direttiva `*ngFor` per visualizzare i dati di ogni stanza, e progettare un template simile a questo:

```
<div class="lucistanze">
  <ul>
    <li *ngFor="let lucisingole of lucidb">
      <i *ngIf="lucisingole.stato == 'ON'" class="material-icons on">
        lightbulb_outline </i>
        {{lucisingole.stanza}}
      </li>
    </ul>
  </div>
```

Come puoi osservare, all'interno del tag ``, ho inserito la notazione `*ngIf`, e all'interno delle doppie virgolette, un'espressione che rappresenta la condizione da verificare:

```
lucisingole.stato == 'ON'
```

Il simbolo di doppio uguale `==`, fa parte degli operatori di confronto di TypeScript/JavaScript, come ben saprai.

A seconda del risultato dell'uguaglianza, “vero” o “falso”, il corrispondente tag `<i>` con associata la direttiva `*ngIf`, verrà o meno visualizzato.

Il codice completo dell'ipotetico componente `lucicasa.component.ts`, con selettore `<ca-lucicasa>`, potrebbe essere scritto in questo modo:

```
lucicasa/lucicasa.component.ts

import { Component } from '@angular/core';
@Component({
  selector: 'ca-lucicasa',
  template:
`<div class="lucistanze">
  <ul>
    <li *ngFor="let lucisingole of lucidb">
```

(continua)

```

<i *ngIf="lucisingole.stato == 'ON'" class="material-icons on">
    lightbulb_outline
</i> {{lucisingole.stanza}}
</li>
</ul>
</div> `,
styleUrls: ['./lucicasa.component.css']
})
export class LucicasaComponent {
    lucidb = [{ stanza: 'Luce Cucine',
        stato: 'OFF',
        luminosita: 5
    },
    { stanza: 'Luce Sala Pranzo',
        stato: 'ON',
        luminosita: 2
    },
    { stanza: 'Luce Bagno',
        stato: 'OFF',
        luminosita: 8
    }];
}

```

dove abbiamo inserito dei dati di test all'interno di un oggetto JSON, mentre le regole CSS per modifica re lo sfondo dell'icona, le abbiamo inserite all'interno di un file *lucicasa.component.css*, specifico del componente.

Il codice CSS, potrebbe contenere all'interno una regola di nome `.on`, che imposta il colore di sfondo dell'elemento:

```

.on {
    background-color: yellow;
}

```

In questo caso abbiamo sfruttat o la proprietà `stato` definita all'interno dell'oggetto JSON, per impostare una condizione del tipo:

```
lucisingole.stato == 'ON'
```

che risulterà essere vera, solo se la relativa proprietà memorizzata nei diversi oggetti, risulta pari alla stringa “ON”.

4.8 Esempio d'uso del blocco else

A partire da Angular 4, la direttiva `*ngIf` è stata potenziata, ed è possibile utilizzare un costrutto molto simile alla classica condizione `if, else` di molti linguaggi di programmazione.

Questo ci può essere d'aiuto per visualizzare il simbolo della lampada dina, anche nella condizione in cui sia spenta.

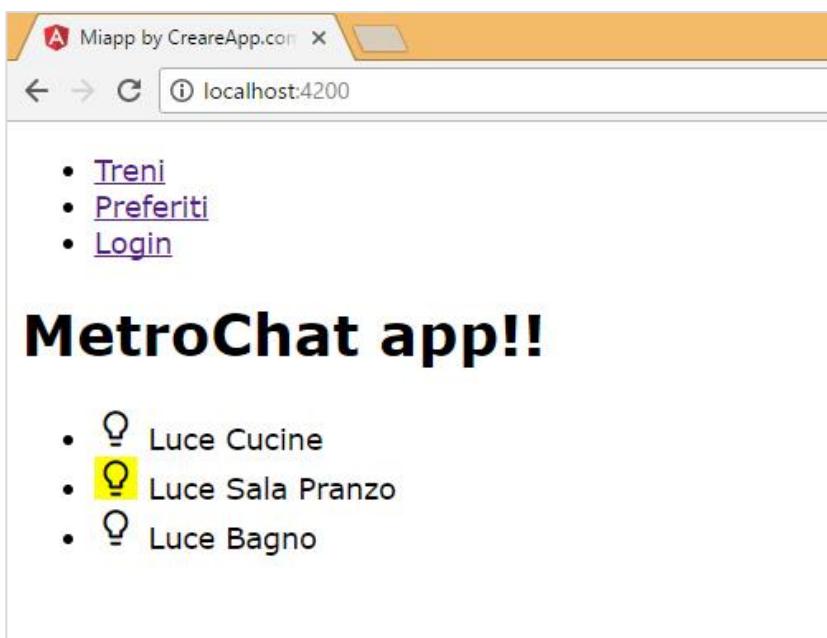


Figura 4.5 – Visualizzazione condizionata di un'icona

Per fare questo si può sfruttare questa notazione:

```
<taghtml *ngIf="condizione; else secondoblocco">...</taghtml>
<ng-template #secondoblocco>...</ng-template>
```

Se la condizione è “vera”, sarà visualizzato il tag `<taghtml>`, mentre se “falsa”, verrà visualizzato un secondo blocco di codice che dovrà essere racchiuso all'interno della direttiva `<ng-template>` a cui è stata associata la variabile del template locale `#showluce`, affinché sia identificata.

```
<div class="lucistanza">
  <ul>
    <li *ngFor="let lucisingole of lucidb">
      <span *ngIf="lucisingole.stato == 'OFF'; else showluce">
        <i class="material-icon">lightbulb_outline</i>
        {{lucisingole.stanza}}
      </span>
      <ng-template #showluce>
        <span>
          <i class="material-icon on">lightbulb_outline</i>
          {{lucisingole.stanza}}
        </span>
      </ng-template>
    </li>
  </ul>
</div>
```

Non appena impareremo a gestire gli eventi, vedremo come implementare anche delle funzionalità che ci permetteranno di accendere le lampadine di una stanza, con un click sopra alla relativa icona.

Come dicevo, tutte queste nozioni che ci serviranno per lo sviluppo dell'applicazione MetroChat, quando dovremo inserire tra i preferiti un determinato messaggio chat, cliccando sulla relativa icona.

Altri esempi li puoi vedere consultando la pagina ufficiale:
<https://angular.io/api/common/NgIf>

4.9 Condizioni multiple: direttiva ***ngSwitch**

Grazie alla direttiva strutturale ***ngIf**, riusciamo a visualizzare un determinato elemento del DOM, sulla base del risultato - “vero” o “falso” - di un'espressione.

In altri casi, potresti avere la necessità di effettuare una scelta sulla base di più valori possibili, non solo “vero” o “falso”.

Per questo, ci viene in aiuto la direttiva **ngSwitch**, che dovrà essere usata con la seguente sintassi:

```
<tagContenitore [ngSwitch]="espressione">
  <tagFiglio *ngSwitchCase="valoreA">AAA</tagfiglio>
  <tagFiglio *ngSwitchCase="valoreB">BBB</tagfiglio>
  <tagFiglio *ngSwitchCase="valoreC">CCC</tagfiglio>
</tagContenitore>
```

NB: Nota come `ngSwitch`, sia stata inserita all'interno delle **parentesi quadre**. Questa notazione ti diventerà familiare ed è stata adottata dal team di Angular, per collegare una proprietà di un componente o di un elemento HTML del DOM a dei dati o a un'espressione presente alla destra dell'uguale. Questa tecnica viene usata in Angular per inizializzare un elemento o impostare uno stato per una direttiva, come nel nostro caso.

Questa direttiva si applica a strutture HTML che presentano un tag genitore e tanti figli. L'espressione inserita alla destra dell'uguale, sarà valutata, e sulla base del valore restituito, verrà mostrato uno dei tag figlio.

E' chiaro che sarà visualizzato solo il tag figlio corrispondente al valore inserito all'interno della direttiva `*ngSwitchCase`.

Quest'ultima può essere inserita anche sfruttando la notazione con le parentesi quadre, ossia scrivendo: **[ngSwitchCase]**

Riprendendo l'esempio dell'ipotetico applicativo per la gestione da remoto della nostra abitazione, se all'interno della proprietà `luminosità`, avessi un valore che rappresenta l'intensità della luce attualmente impostata in una stanza, da 1 a 5, allora potresti mostrare all'utente una lampadina di colore diverso, dal nero (buio), al giallo paglierino (tanta luce).

Provando per ora a inserire questa informazione sotto forma di stringa:

```
<h1>Luce proveniente dall'esterno</h1>
<div [ngSwitch]="luminosità">
  <span *ngSwitchCase="1" class="buio">Buio</span>
  <span *ngSwitchCase="2" class="penombra">Molto Nuvoloso</span>
  <span *ngSwitchCase="3" class="nuvoloso">Nuvoloso</span>
  <span *ngSwitchCase="4" class="solenuvole">Sole e Nuvole</span>
  <span *ngSwitchCase="5" class="sole">Sole</span>
</div>
```

Nel caso in cui il valore impostato per la variabile `luminosità` fosse 2, allora verrebbe visualizzato il tag con l'attributo di classe impostato a " `penombra`" e la scritta "Molto Nuvoloso".

Nel caso in cui il valore della variabile non corrisponda ai valori da 1 a 5, non verrà visualizzato alcun tag figlio.

In alternativa è possibile usare una terza direttiva `*ngSwitchDefault`, che ti permetterà di visualizzare sempre almeno un tag figlio, nell'ipotesi le condizioni non siano soddisfatte dai precedenti `ngSwitchCase`.

Potrebbe succedere infatti che l'applicazione non sia in grado di leggere il dato remoto, fornito dal sensore. In questo caso potresti visualizzare comunque un messaggio informativo all'utente:

```
<h1>Luce proveniente dall'esterno</h1>
<div [ngSwitch]="luminosità">
  <span *ngSwitchCase="1" class="buio">Buio</span>
  <span *ngSwitchCase="2" class="penombra">Molto Nuvoloso</span>
  <span *ngSwitchCase="3" class="nuvoloso">Nuvoloso</span>
  <span *ngSwitchCase="4" class="solenuvole">Sole e Nuvole</span>
  <span *ngSwitchCase="5" class="sole">Sole</span>
  <span *ngSwitchDefault class="errore">Errore Lettura</span>
</div>
```


Capitolo 5

Cambiare lo stile di elementi del DOM

5.1 Aggiungere o togliere proprietà CSS con `ngStyle`

Entriamo nel vivo delle “**direttive attributo**”, in grado di cambiare l’aspetto o il comportamento di un elemento del DOM.

Abbiamo già sottolineato che in Angular , il collegamento tra una proprietà di un elemento del DOM e i dati, avviene grazie alla notazione delle doppie parentesi quadre - [] -, che ci permette di aggiungere delle direttive *attributo*, come attributi di un elemento, con la stessa notazione usata nei CSS per i selettori di attributo.

Nel mondo delle pagine web, i fogli di stile o CSS , come ben sai, sono l’elemento base con cui andare a modificare l’aspetto di ogni elemento di una pagina, dal testo, alla disposizione di immagini, ai colori etc. Anche in Angular posso sfruttare i fogli di stile per modificare l’aspetto di ogni elemento definito all’interno del template.

Oltre a questo, posso avvermi anche di alcune direttive che vanno a modificare l’**aspetto di un componente**, senza che manualmente debba andare a inserire le diverse regole CSS nel template. La prima di queste, agisce direttamente sull’attributo *style* di un tag html.

Vediamo subito un esempio. Se volessi cambiare **dinamicamente** la dimensione del un testo definito qui sotto, come potesti fare?

```
<p style="font-family:11px">Questa è una news con il testo molto  
piccolo</p>
```

Devi avere la possibilità di iniettare un valore diverso da 11px all'interno del tag `p` della pagina. Si ottiene questo con la direttiva `ngStyle`.

notizie/news.component.ts

```
import { Component } from '@angular/core';
@Component({
  selector: 'ca-news',
  template: `<p [ngStyle]="miostile">
    News con il testo molto piccolo
  </p>`
})
export class NewsComponent {
  miostile: object;
  constructor() {
    this.miostile = {'font-size': '14px'};
  }
}
```

L'effetto che otterrai, sarà un ingrandimento del carattere a 14 px.

Come vedi, all'interno del tag `p` ho inserito un attributo con la classica notazione delle parentesi quadre, che mi permetterà di valorizzare la proprietà `style` del tag HTML con un oggetto JavaScript, la cui prima chiave è proprio la proprietà CSS che si vuole impostare.

NB: Nota come `miostile` sia un oggetto “Literal JavaScript”, dove la chiave corrisponde alla proprietà CSS che si vuole impostare, il tutto racchiuso all'interno di singole virgolette. In realtà per alcune proprietà potresti farne a meno, ma per non sbagliare è meglio inserirle sempre.

Se oltre a cambiare la dimensione, volessi cambiare anche la colorazione del testo, allora dovrei valorizzare la variabile `miostile` con il seguente oggetto JS:

```
// ho aggiunto due proprietà CSS
this.miostile = {'font-size': '14px', 'color': 'red'};
```

Una caratteristica della direttiva `ngStyle`, è che eventuali attributi di stile applicati direttamente nel template, non vengono persi ma si aggiungono a quelli applicati con la direttiva. Se, ad esempio, il testo avesse una tipografia di carattere “Arial” applicata tramite una regola CSS “inline”:

notizie/news.component.ts

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

(continua)

```

    selector: 'ca-news',
    template: `<p style="font-family: Arial" [ngStyle]="miostile">
      News con il testo molto piccolo
    </p>`
  }

export class NewsComponent {
  miostile: object;
  constructor() {
    this.miostile = {'font-size': '14px', 'color': 'red'};
  }
}

```

il risultato finale che visualizzerò nel DOM, sarà un testo con carattere “Arial”, dimensione 14 px, colorato di rosso.

Chiaramente potresti inserire direttamente nel template alcune proprietà che ritieni non debbano cambiare, e lasciare quelle variabili nella direttiva; oppure aggiungere tutto nella direttiva, esplicitando le sezioni modificabili, con il nome di una variabile da valorizzare nel corpo della classe.

notizie/news.component.ts

```

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

@Component({
  selector: 'ca-news',
  template: `<p [ngStyle]="{{'font-family': 'Arial', 'color':'red',
  'font-style': dimcarattere}}>News con il testo molto piccolo</p>`
})

export class NewsComponent {
  dimcarattere: string;
  constructor() {
    // dimensione carattere con unità di misura
    this.dimcarattere = '14px';
  }
}

```

L'oggetto che riceve la direttiva `ngStyle`, accetta la scrittura di proprietà CSS, con due tipologie di notazione: "kebab-case" (quella usata fino ad ora) o "camel case" ossia eliminando il trattino (-) da quei nomi che li presentano e iniziando la seconda parte della parola in maiuscolo.

Meglio a farsi che a dirsi. Ecco alcuni esempi di trasformazione di proprietà CSS con questa notazione:

- padding-top > paddingTop
- border-bottom > borderBottom

e così via.

5.2 Aggiungere o togliere regole CSS

Sempre nell'ambito delle modifiche all'aspetto di un elemento o di un tag html presente nel DOM, sappiamo che con gli attributi CSS di classe, è possibile applicare una regola o insieme di regole a più tag html.

Ad esempio, il paragrafo qui sotto ha impostato un attributo di classe "news", che sarà definito come regola nel foglio di stile.

```
<p class="news">Testo di una notizia, formattata con i CSS</p>
```

Ora se il mio obiettivo è quello di andare a modificare dinamicamente l'aspetto di alcuni elementi della pagina bypassando la direttiva `ngStyle`, l'alternativa è usare la direttiva `ngClass`.

Una delle peculiarità di quest'ultima direttiva è che può essere utilizzata con stringhe, array, oppure oggetti.

5.3 Uso di `ngClass` con una proprietà stringa

L'esempio è molto simile a quello visto per l'altra direttiva `ngStyle`. Si dovrà “incastrare” la direttiva all'interno delle parentesi quadre, e il valore dovrà essere impostato come se fosse una stringa rappresentativa delle diverse classi da aggiungere a quelle già eventualmente presenti.

notizie/news.component.ts

```
@Component({
  selector: 'ca-news',
  template: `<p [ngClass]="mieclassi" class="big-text">
    News con testo molto piccolo
  </p>`,
  styleUrls: ['./miostile.css']
})
export class NewsComponent {
  mieclassi:string;
  constructor() {
```

(continua)

```

    this.mieclassi = 'text-center light-blue';
}
}

```

In questo caso, ho usato un file esterno `miostile.css`, con all'interno definire le regole `light-blue`, `text-center`, e `big-text`.

```

.big-text{
  font-size:18px;
}
.text-center{
  text-align:center;
}
.light-blue{
  color:#ADD8E6;
}

```

Le classi definite all'interno della proprietà `mieclassi`, verranno aggiunte a quella già presente e definita nel template (`big-text`).

In questo modo posso modificare dinamicamente lo stile di specifiche sezioni del template, usando nomi di regole definite nel file CSS.

5.4 Uso di `ngClass` con un array

Spesso capita che i valori da aggiungere dinamicamente, siano recuperati da una sorgente esterna che li confeziona all'interno di un array. In Angular è possibile usare anche la seguente notazione:

```

notizie/news.component.ts

//...
@Component({
  selector: 'ca-news',
  template: `<p [ngClass]="mieclassi" class="big-text"></p>`,
  styleUrls: ['./miostile.css']
})

export class NewsComponent {
  mieclassi:Array<string>;
  constructor(){
    // applico due attributi di classe
    this.mieclassi = ['text-center', 'light-blue'];
  }
}

```

5.5 Uso di ngClass con un oggetto

Una caratteristica interessante della direttiva `ngClass`, è che posso sfruttarla non solo per aggiungere attributi di classe come visto fino ad ora ma anche per togliere attributi di classe, sulla base di un valore booleano VERO o FALSO.

Questo è possibile solo sfruttando la notazione qui sotto, che prevede l'inserimento di un oggetto JavaScript le cui chiavi sono rappresentate dagli attributi di classe, mentre il valore è rappresentato da un dato booleano, impostato a "true" o "false" a seconda che tu voglia aggiungere o eliminare quel particolare attributo.

```
this.isCentered = true;
this.isBlue= false;
this.mieclassi = {'text-center' : this.isCentered,
                 'light-blue': this.isBlue};
```

Questo, ad esempio, può essere usato per mostrare o oscurare un determinato tag html della pagina.

notizie/news.component.ts

```
// ...
@Component({
  selector: 'ca-news',
  template: `<p [ngClass]="mieclassi" class="big-text"></p>`,
  styleUrls: ['./miostile.css']
})

export class NewsComponent {
  mieclassi: object;
  isCentered: boolean;
  isBlue: boolean;
  constructor() {
    this.isCentered = true;
    this.isBlue= false;
    this.mieclassi = {'text-center' : this.isCentered, 'light-blue': this.isBlue};
  }
}
```

In questo caso, il risultato finale non mostrerà l'attributo di classe `light-blue`, in quanto è stato oscurato dal valore impostato per la variabile `isBlue`:

```
<p class="text-center big-text">
```

Capitolo 6

Formattare i dati con i PIPE

6.1 Formattare le date e l'ora

La visualizzazione dei dati nel template, può essere arricchita di uno strumento molto potente che viene chiamato “Pipe”.

Per la nostra applicazione MetroChat, sfrutteremo un Pipe personalizzato che svilupperemo ad hoc.

Chiaramente in Angular vi sono già diversi Pipe sviluppati, che posso sfruttare sia per formattare delle date, sia per formattare del testo.

Il nome deriva dal fatto che si fa seguire il simbolo di pipe |, al nome della proprietà che si vuole visualizzare nel template con la tecnica dell'interpolazione:

```
 {{ nomeproprietà | NOME_PIPE: PARAMETRO }}
```

Ad esempio, la data di un giorno può essere rappresentata in diversi modi, pertanto potrebbe essere utile avere uno strumento che al volo, mi permetta di cambiare il formato senza dover scrivere, di volta in volta, del codice nella classe.

Nel caso di una data, sarà necessario usare il "DatePipe" che ha questa notazione:

```
 {{ datadaformattare | date: PARAMETRO }}
```

dove `datadaformattare` è una variabile di tipo `Date`.

Ad esempio, nella seguente notazione:

```
// visualizza la data nel formato: MM/GG/AA
{{ datadaformattare | date:'shortDate' }}
```

visualizzo la data nel formato Mese, Giorno e Anno a 2 cifre (es. 12/01/22, ossia il primo dicembre 2022).

Altre notazioni di uso comune sono:

- 'shortDate': equivalente a MM/GG/AA (Es. 12/25/22)
- 'fullDate': equivalente a GiornoSettimana, Mese GG, AAAA
- 'longDate': equivalente a Mese GG, AAAA

Ognuna di queste può essere creata anche sfruttando la notazione con lettere rappresentative del giorno, mese e anno, come indicato qui sotto:

- d: numero indicativo del giorno (1 cifra senza zero. Es. 1, 9, 15)
- M: numero indicativo del mese (deve essere maiuscolo. Es. 9,12)
- y: numero minimo cifre indicative dell'anno (es. 4 cifre)
- MMMM: nome del mese per esteso (es. June)
- MMM: nome del mese abbreviato a tre lettere (es. Jun)
- EEEE: nome del giorno della settimana (es. Monday)
- EEE: nome del giorno della settimana abbreviato (es. Mon)

Quindi, per ottenere la stessa data indicata sopra con mese, giorno e anno:

```
// NB: le ho invertite per avere MM/GG/AAAA
{{ datadaformattare | date:'Mdy' }}
```

In alternativa, avrei potuto mantenere lo stesso ordine, inserendo manualmente il simbolo di /:

```
{{ datadaformattare | date:'M/d/y' }}
```

Nel caso di una misura del tempo, posso sfruttare le lettere:

- h: ora
- m: minuti
- s: secondi

Riassumendo, nell'ipotesi volessi formattare una data creata con la classica istruzione:

```
data = new Date();
```

potrei scrivere:

pipe/test.pipe.ts

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

@Component({
  selector: 'ca-pipe',
  template:
    <h3>Esempio di PIPE con le Date</h3>
    <p>{{dataoggi | date:'shortTime'}}</p>
    <p>{{dataoggi | date:'mediumDate'}}</p>
    // mostro GG Mese AAAA
    <p>{{dataoggi | date:'d MMMM y'}}</p>
    // mostro l'ora HH:MM:SS
    <p>{{dataoggi | date:'hms'}}</p>
})
export class TestPipe {
  dataoggi: Date;
  constructor() {
    this.dataoggi = new Date();
  }
}
```

6.2 Formattare stringhe e numeri

Proseguendo la panoramica sulle tipologie di pipe predefinite, vediamo quelle che ci permettono di trasformare un testo in maiuscolo. Si tratta del pipe "UppercasePipe".

La sintassi è:

```
{{ stringa | uppercase }}
```

Analogo discorso per il minuscolo:

```
 {{ stringa | lowercase }}
```

Nel caso di numeri frazionari, spesso si ha la necessità di rappresentarli con un certo numero di decimali. La sintassi sarà:

```
 {{ numero | number: 'PARAMETRO' }}
```

dove parametro deve essere costruito con questa sintassi:

```
 PARAMETRO = Minimo-num-cifre-intere . Minimo numero cifre decimali -
Massimo numero cifre decimali
```

Ad esempio, usando la notazione:

```
 2.1-2 // Es. 12.43, 12.4, 08.3
```

intendo un numero rappresentato da due cifre intere, e da 1 a 2, cifre decimali. Se il numero fosse costituito da più di due cifre decimali, avverrebbe l'arrotondamento automatico.

Applicando il pipe alla variabile numero valorizzata a 1.348 otterrei:

```
 {{ numero | number: '2.1-2' }} // ottengo 01.35
```

Analogo discorso per un numero percentuale, che necessiterà dell'uso del pipe "PercentPipe" con la seguente sintassi:

```
 {{ numero | percent: 'PARAMETRO' }}
```

dove la notazione : 'PARAMETRO' in questo caso è opzionale ma se inserito, segue la stessa logica vista per un numero.

Ad esempio, se il numero da trasformare in percentuale fosse 0.3267 e scrivessi:

```
{{ numero | percent }} // ottengo: 32.67%
```

Se invece aggiungessi il parametro:

```
{{ numero | percent: '2.1-1' }} // ottengo: 32.7%
```

Altro pipe che spesso viene usato in fase di test per visualizzare un oggetto JSON, è dato dal pipe “JsonPipe”, che equivale ad utilizzare il metodo `JSON.stringify()`, come impareremo a fare quando parleremo di invio e ricezione dati via HTTP.

Se avessimo un oggetto JavaScript del tipo:

```
this.metro = {'idt':'12345', 'linea': 'Rossa'};
```

e provassimo a visualizzarlo nel template, otterremmo solo una stringa [object Object], mentre se lo scrivessimo nella forma:

```
 {{ metro | json }}
```

otterremmo la stessa visualizzazione di come è rappresentato nella classe.

6.3 Creare Pipe personalizzati

Dopo aver visto alcuni dei pipe più usati in Angular, vediamo come si possa creare un nostro pipe personalizzato che ci servirà all'interno del l'applicazione MetroChat, per rappresentare i minuti e secondi mancanti alla partenza di un treno , con una notazione del tipo:

```
 {{ attesa | mmss }}
```

L'ipotesi che avevamo fatto infatti, quando abbiamo visto come rappresentare la sequenza di treni in arrivo grazie al componente `TreniComponent`, era che l'orario di partenza fosse espresso in millisecondi.

Per conoscere il tempo mancante alla partenza, dovremo fare la differenza tra l'ora attuale espressa in millisecondi e quella di partenza, per poi trasformare il tutto in minuti e secondi e rappresentarli in un formato comprensibile al volo dall'utente.

In JavaScript, la funzione `getTime()` ci permette di ottenere l'ora attuale espressa in millisecondi a partire dalla famosa data di Unix (01/01/1970). Ipotizzando pertanto che la data di partenza sia anch'essa espressa nello stesso formato, dovrei inserire le semplici operazioni qui sotto per avere i minuti e secondi:

```
attesa = orapartenza-oraattuale;
minuti = Math.floor(attesa / 60000);
secondi = Math.floor((attesa - this.minuti*60000)/1000);
```

dove ho diviso per $60*1000$ per avere il numero di minuti, arrotondati con il metodo `floor()`, e sottratto questi al tempo di attesa, per ottenere la parte mancante in secondi.

A questo punto dobbiamo mostrare i due dati nella forma MM:SS dove con MM intendo i minuti e con SS i secondi. Chiaramente devo anteporre degli zeri per valori inferiori a 10, quindi potrei avvalermi di una semplice funzione di appoggio che sfrutta un array di 4 elementi, valorizzato con il numero a una o due cifre, per poi traslare le prime due posizioni dell'array verso sinistra.

```
duecifre(numero:number,zero:string,length:number):string {
    return (new Array(length+1).join(zero)+numero).slice(-length);
}
```

ottenendo così:

```
attesa = orapartenza-oraattuale;
minuti = Math.floor(attesa / 60000);
secondi = Math.floor((attesa - this.minuti*60000)/1000);
contorovescia = duecifre(minuti,'0',2)+':'+ duecifre(secondi,'0',2);

duecifre(numero:number,zero:string,length:number):string {
    return (new Array(length+1).join(zero)+numero).slice(-length);
}
```

Queste righe, possono diventare la base per creare un pipe personalizzato, che potremmo chiamare **mmss**, da usare nel template del componente *TreniComponent*.

Per creare un pipe, possiamo avvalerci della linea di comando, che offre il vantaggio di aggiungere già gran parte delle righe che costituiscono un pipe.

Per mantenere ordine tra le diverse cartelle, posso creare una cartella apposita di nome *pipe* all'interno della quale ci sposteremo per eseguire la linea di comando:

```
ng g pipe mmss
```

Verrà generato un nuovo file dal nome *mmss.pipe.ts*, che sarà aggiunto in automatico all'interno del file *app.module.ts*, al fine di renderlo disponibile per tutti i componenti, senza necessità di importare alcuna riga aggiuntiva.

```
app.module.ts

import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { MiaAppComponent } from './app.component';
import { MenuComponent } from './menu/menu.component';
import { MmssPipe } from './pipe/mmss.pipe';

@NgModule({
  imports: [ BrowserModule ],
  declarations: [ MiaAppComponent, MenuComponent, MmssPipe ],
  bootstrap: [ MiaAppComponent ]
})

export class AppModule {}
```

Il file *mmss.pipe.ts* così creato, avrà questa struttura:

```
pipe/mmss.pipe.ts

import { Pipe, PipeTransform } from '@angular/core';

@Pipe({
  name: 'mmss'
})

export class MmssPipe implements PipeTransform {
  transform(value: any, args?: any): any {
    return null;
  }
}
```

Come puoi osservare si tratta di una classe, a cui è stato aggiunto un decoratore `@Pipe()` con una proprietà `name`, pari al nome scelto per il pipe.

La classe implementa `PipeTransform`, con il metodo `transform` che dovrà essere sviluppato. I parametri ricevuti in ingresso sono il dato aggiunto nel template, e dei dati opzionali, così come visto per altri pipe predefiniti, con la notazione `nomepipe:'PARAMETRI'`

Nel nostro caso, non passeremo dei parametri aggiuntivi, ma solo il dato legato alla differenza tra il tempo di partenza e il tempo attuale espresso in millisecondi. Sfruttando pertanto le righe già sviluppate:

```
pipe/mmss.pipe.ts

import { Pipe, PipeTransform } from '@angular/core';
@Pipe({
  name: 'mmss'
})
export class MmssPipe implements PipeTransform {
  minuti!: number;
  secondi!: number;
  contorovescia!: string;
  transform(attesa: number, args?: any): string {
    this.minuti = Math.floor(attesa/60000);
    this.secondi = Math.floor((attesa - this.minuti*60000)/1000);
    this.contorovescia = this.duecifre(this.minuti,'0',2)+':'
      + this.duecifre(this.secondi,'0',2);
    return this.contorovescia;
  }
  duecifre(numero:number,zero:string,length:number):string {
    return (new Array(length+1).join(zero)+numero).slice(-length);
  }
}
```

Il valore restituito da `transform` è una stringa. Questo pipe vedremo ci servirà per rappresentare l'orario di partenza sotto forma di **conto alla rovescia** non appena avremo sviluppato i primi componenti "intelligenti".

Pertanto se la proprietà `attesa` fosse valorizzata con '123400' otterrei: 02:03

Attesa: 123400

Attesa con pipe: 02:03

Figura 6.1 – Effetto dell'applicazione di un pipe personalizzato sull'ora

Capitolo 7

Modellare i dati

7.1 Una classe per definire il tipo di dati

Fino ad ora abbiamo visto che i dati visualizzati all'interno del template di un componente, venivano inseriti direttamente all'interno del corpo della classe, valorizzando una variabile.

Questo può andare bene per fare dei piccoli test ma nelle applicazioni reali, avrai la necessità di costruire delle rappresentazioni dei dati, che siano facili da gestire e da mantenere aggiornati, oltre che essere coerenti con il tipo di informazione da memorizzare in ognuno.

Se poi pensiamo che la maggior parte dei dati che un'applicazione visualizzerà, arriveranno da una sorgente esterna, tipicamente sotto forma di oggetti JSON, dobbiamo trovare un modo per maneggiarli senza introdurre potenziali errori.

In previsione di dover gestire dei dati di un certo “TIPO” con la possibilità di manipolarli, il pensiero va subito alle classi.

L'idea è quella di definire una classe modello, che conterrà all'interno la definizione dei diversi campi in grado di rappresentare una particolare tipologia d'informazione.

In sostanza si crea un nuovo tipo di dati, che potremmo usare esattamente come si fa per un qualsiasi tipo di dati primitivo (numero, stringa, etc.).

Riprendendo il classico esempio delle notizie, costituite da un titolo, una descrizione e da un identificativo numerico per accedere all'eventuale dettaglio, queste potrebbero essere visualizzare all'interno del template del componente <ca-listanews>, con la classica direttiva *ngFor:

• Titolo Notizia 1

Descrizione Notizia 1

• Titolo Notizia 2

Descrizione Notizia 2

• Titolo Notizia 3

Descrizione Notizia 3

• Titolo Notizia 4

Descrizione Notizia 4

Figura 7.1 – Elenco notizie da modellare con una classe o interfaccia

Ogni riga rappresenta un'informazione che potrete modellare definendo la classe di nome News, da memorizzare nel file *news.model.ts* in questo modo:

```
export class News {
  constructor(public id: number, public titolo: string, public desc: string) {
  }
}
```

Ogni qualvolta definiamo una classe in TypeScript comprensiva di costruttore, nella realtà avvengono multiple dichiarazioni allo stesso istante.

I tre parametri di nome `id`, `titolo` e `descrizione`, con relativo **tipo di dati**, grazie alle peculiarità di TypeScript, sono a tutti gli effetti delle proprietà della classe definite con lo stesso nome.

E' proprio il compilatore TypeScript che crea una proprietà pubblica per ogni parametro presente nel costruttore, assegnandogli il valore non appena viene creato un nuovo oggetto con l'operatore `new`.

E' sempre buona norma separare i diversi pezzi che costituiscono l'applicazione, quindi ti consiglio di creare una cartella di nome *model*, in cui andare a salvare le diverse rappresentazioni dei dati.

I file tipicamente vengono salvati aggiungendo al nome identificativo della classe, l'estensione *.model.ts*

Nota la presenza della parola `export`, necessaria al fine di esportare la classe come modulo e poterla poi importare nei diversi componenti che avranno la necessità di gestire questo tipo di dati.

Per ogni parametro del costruttore ho aggiunto la parola `public`, anche se avrei potuto ometterla, in quanto TypeScript considera le proprietà senza indicazione sulla visibilità, sempre di tipo `public`

La notazione TypeScript abbreviata qui sopra, sostanzialmente equivale a scrivere:

```
export class News {
    // variabili di classe
    id: number;
    titolo: string;
    desc: string;

    constructor(id:number, titolo:string, desc:string) {
        this.id = id;
        this.titolo = titolo;
        this.desc = desc;
    }
}
```

I puristi della programmazione ad oggetti, potrebbero storcere il naso, in quanto definire pubbliche proprietà di una classe è sempre rischioso, ma a 1 limite possono sempre creare dei **getter** e **setter** e rendere le proprietà private.

Come ogni classe, potresti non limitarti solo alla definizione di proprietà ma definire anche dei metodi, che andranno ad effettuare delle operazioni specifiche sui dati ricevuti.

Limitandoci per ora alle proprietà della classe, nel caso volessi creare e valorizzare un nuovo dato di tipo `News`, potresti sfruttare la classica notazione con l'operatore `new`, e inserire nel componente che necessita di usare questo tipo di dati, le seguenti righe:

```
// Array di oggetti News
listanew: News[] = [];
constructor() {
  this.listanew.push(
    new News(1, 'Titolo Notizia 1', 'Descrizione notizia 1')
  );
}
```

Come vedi, ho passato al costruttore della classe News, tre parametri, che sono i dati della prima notizia da visualizzare. Questo oggetto è stato poi inserito all'interno dell'array listanew, che conterrà proprio una lista di oggetti di tipo News.

E' chiaro che Angular non sa come rappresentare l'oggetto News, quindi dobbiamo indicargli dove si trova la relativa definizione, sfruttando l'importazione della classe. Pertanto il componente <ca-listanews>, progettato nelle precedenti lezioni, diventerebbe:

```
notizie/notizie.component.ts

import { Component } from '@angular/core';
import { News } from './model/news.model';

@Component({
  selector: 'ca-listanews',
  template:
    <div class="news">
      <ul>
        <!--elemento che si ripete N volte-->
        <li *ngFor="let notizia of listanews">
          <h2>{{notizia.titolo}}</h2>
          <p>{{notizia.desc}}</p>
        </li>
      </ul>
    </div>`
```

-)

```
export class NotizieComponent {
  // Array di oggetti News
  listanews: News[] = [];
  constructor () {
    this.listanew.push(
      new News(1, 'Titolo Notizia 1', 'Descrizione notizia 1')
    );
    this.listanew.push(
      new News(2, 'Titolo Notizia 2', 'Descrizione notizia 2')
    );
    this.listanew.push(
      new News(3, 'Titolo Notizia 3', 'Descrizione notizia 3')
    );
  }
}
```

Abbiamo sicuramente fatto un passo in avanti a livello di "pulizia" del codice, in quanto ora `News` è un tipo di dati definito esternamente alla classe, e tutti gli usi interni ai diversi componenti, rispetteranno la stessa struttura di informazioni da memorizzare, oltre che il tipo.

Nonostante questo, i valori delle notizie sono ancora presenti inseriti all'interno del corpo della classe, e se volessi usare gli stessi valori per mostrarli in un altro componente, dovrei manualmente ricopiare le stesse righe.

Non preoccupiamoci per ora, perch é vedremo in seguito che la separazione dei dati dal componente, si farà grazie alla creazione di quello che prende il nome di "Service", che sarà iniettato all'occorrenza all'interno del componente che lo richiederà.

7.2 Il modello dati per l'applicazione MetroChat

Sulla base delle nozioni viste, possiamo progettare anche il modello dati per rappresentare gli oggetti che costituiranno il cuore dell'applicazione che intendiamo progettare ossia l'oggetto `Metro`, identificativo del treno in arrivo, e l'oggetto `Messaggio`, identificativo del messaggio chat scambiato tra i diversi utenti presenti all'interno di un treno.

Entrambi potrebbero essere salvati all'interno della cartella `model` creata in precedenza, sempre interna alla cartella `app`.

Una possibile struttura per l'oggetto `Metro`, da memorizzare all'interno del file `metro.model.ts`, potrebbe essere:

```
model/metro.model.ts

export class Metro {
  constructor (
    public idt: string,
    public linea: string,           // nome della linea
    public numchatting: number,     // numero passeggeri in chat
    public tempo: number            // orario partenza
  ) {}
}
```

Altri dati li aggiungeremo più avanti, quando vedremo come recuperare le informazioni da sorgenti esterne via HTTP.

Una possibile struttura invece per l'oggetto `Messaggio` - `messaggio.model.ts` - potrebbe essere:

model/messaggio.model.ts

```
export class Messaggio {
    constructor (
        public idm: number,           // identificativo del messaggio
        public idt: string,          // identificativo del treno
        public idu: string,          // identificativo dell'utente
        public testo: string,         // testo del messaggio
        public idd?: string,         // identif. del destinat. (opzionale)
    ) {}
}
```

La notazione vista per il campo `idd`, la spiegheremo nelle prossime pagine.

7.3 Una classe senza costruttore

La notazione vista in precedenza, ci ha obbligati ad usare l'operatore `new` per creare l'istanza di un nuovo oggetto da inserire all'interno di un array di tipo `News`.

Questo potrebbe essere oneroso a livello di programmazione, quindi quello che spesso si fa, è definire una classe senza costruttore e sfruttare la possibilità offerta da TypeScript, di inizializzare un oggetto assegnando un valore alle diverse proprietà.

Sulla base di queste considerazioni, il modello per rappresentare la notizia può essere riscritto in questo modo:

model/news.model.ts

```
export class News {
    id: number;
    titolo: string;
    descrizione: string;
}
```

e per valorizzare l'array `listanew`, dovrà usare:

```
this.listanew.push({id:1,
                    titolo:'Titolo Notizia1',
                    descrizione:'Descrizione notizia 1'
                });
```

Come puoi osservare, abbiamo aggiunto all'array direttamente un oggetto JavaScript, costruito sulla base della struttura del modello.

Questo perché la maggior parte dei servizi remoti che interrogherò, forniranno i dati sotto forma di array di oggetti JSON.

La classe non ha definito alcun costruttore o metodo interno e non ha la necessità di essere istanziata per essere usata. La si usa in sostanza, solo per definire un nuovo tipo di dati, che sarà il tipo `News`. La stessa cosa si potrebbe fare sfruttando queste due ulteriori notazioni:

```
model/news.model.ts
```

```
export type News = {
  id: number;
  titolo: string;
  descrizione: string
}
```

```
export interface News {
  id: number;
  titolo: string;
  descrizione: string
}
```

con la differenza che in questo caso non posso creare dei metodi interni per manipolare le proprietà dell'oggetto. In genere, se non si devono gestire dei metodi, l'ultima notazione (`export interface`) è quella preferita.

Esercitazione: prova a trasformare la struttura da ti `metro.model.ts`, in modo che non abbia il costruttore. Trasforma il componente `treni.component.ts`, in modo che la proprietà `listatreni`, sia di tipo `Metro`.

7.4 Simulare dati remoti per semplici test

Per poter effettuare dei test locali al fine di apprendere le caratteristiche di Angular, può essere utile spostare la valorizzazione degli array di dati, in una classe esterna, sotto forma di costante, che avrà cura di importare come se fosse un classico componente.

Pertanto, sfruttando il modello dati creato nell'esercitazione richiesta, potrete creare il file `metroremoti.ts`, internamente alla cartella di nome `dati`, in cui inserire una costante da esportare di nome `LISTAMETRO`:

```
dati/metroremoti.ts
```

```
import { Metro } from './model/metro.model';

export const LISTAMETRO: Metro[] = [
  {idt:'12345', linea:'Rossa', numpass:23, tempo:12500},
  {idt:'67890', linea:'Verde', numpass:23, tempo:145000},
  {idt:'09876', linea:'Gialla', numpass:23, tempo:165000}
];
```

Analogo discorso potrei fare per la lista dei messaggi associati ad una c hat, definendo una costante LISTAMSG, all'interno di un file *messaggiremoti.ts*.

```
dati/messaggiremoti.ts
```

```
import { Messaggio } from './model/messaggio.model';

export const LISTAMSG: Messaggio[] = [
  {idm:1,idt:'12345',idu:'AAA',testo: '1° msg Linea Rossa',idd: ''},
  {idm:2,idt:'12345',idu:'BBB',testo: '2° msg Linea Rossa',idd: ''},
  {idm:3,idt:'67890',idu:'CCC',testo: '3° msg Linea Verde',idd: ''},
];
```

Per poter usare queste costanti – costituite da array di oggetti Metro e Messaggio – all'interno di un componente, dovrò scrivere la classica istruzione di importazione, indicando il percorso relativo da cui prelevare il file:

```
import { LISTAMETRO } from './dati/metroremoti';
import { LISTAMSG } from './dati/messaggiremoti';
```

7.5 Classe con dati opzionali

Negli esempi precedenti abbiamo sempre ipotizzato che tutti i dati definiti nella classe modello, fossero da inserire.

Nell'esempio qui sotto, la classe rappresentativa dell'oggetto News, è costituito da tre dati obbligatori.

```
model/news.model.ts
```

```
export class News {
  id: number;
  titolo: string;
  desc: string
}
```

Non sempre è così, nel senso che alcuni di questi potrebbero essere anche opzionali.

In TypeScript questi ultimi vengono indicati usando l'operatore “punto di domanda” “?”, come nell'esempio qui sotto:

model/news.model.ts

```
export class News {
  id: number;
  titolo: string;
  desc: string;
  datapubb?: string      // proprietà della classe opzionale
}
```

Il membro della classe `datapubb`, risulta così essere un parametro opzionale, e il compilatore accetta che venga creata l'istanza della classe, anche in assenza di tale parametro, come nell'esempio qui sotto:

notizie/notizie.component.ts

```
import { Component } from '@angular/core';
import { News } from './model/news.model.ts';

@Component({
  selector: 'ca-listanews',
  template: `
    <div class="news">
      <ul>
        <li *ngFor="let notizia of listanews">
          <h2>{{notizia.titolo}}</h2>
          <p>{{notizia.desc}}</p>
        </li>
      </ul>
    </div>`
})
export class NotizieComponent {
  // Array di oggetti News
  listanews: News[] = [];
  constructor() {
    this.listanews = [{id:1,
                      titolo: 'Titolo Notizia 1',
                      descrizione: 'Descrizione notizia 1'}
    ];
  }
}
```


Capitolo 8

Interagire con il template e l'app: gli Eventi

8.1 Gestire il click o il tocco su un elemento del template

Tra le operazioni che un utente comunemente fa all'interno di un'applicazione, ci sono l'inserimento di dati tramite tastiera, il click per cambiare pagina, accedere a informazioni di dettaglio, e così via.

In Angular, ogni evento in grado di essere intercettato dal browser, può essere gestito in modo molto simile a quello che avviene in JavaScript.

Se in JavaScript, per gestire il click su di un elemento del DOM si utilizza la seguente sintassi:

```
<taghtml onclick="azione()">...</taghtml>
```

in Angular, si incarta il tipo di evento da monitorare e gestire, all'interno di parentesi tonde ():

```
<taghtml (click)="azione()">...</taghtml>
```

Come vedi la struttura è abbastanza simile a JavaScript. Cambia solo il nome dell'evento del browser, che perde il prefisso "on".

La sintassi da usare in Angular, quindi sarà sempre del tipo:

```
(nomeEVENTO)="gestoreEvento()"
```

Il gestore dell'evento, è una funzione che verrà richiamata non appena si verifica quel particolare evento sull'elemento del DOM, evento in grado di essere intercettato dal browser.

Nell'ipotesi in cui la vista o template del componente, sia rappresentata dal classico campo di input e dal bottone per l'invio, per intercettare la pressione su quest'ultimo potrei scrivere:

```
<button (click)="inviaDato()">Invia</button>
```

Come vedi è stato inserito il nome dell'evento “click” da monitorare, all'interno di parentesi tonde, e poi a destra dell'uguale si è inserito il nome del gestore/metodo della classe che si prenderà cura di recuperare l'informazione.

Nel corpo della classe chiaramente dovrò definire il gestore dell'evento, chiamato nel nostro esempio `inviaDato()`:

```
class MioComponente {
  constructor() {
    // azioni da eseguire all'inizializzazione del componente
  }
  inviaDato() {
    // metodo per l'invio del dato
  }
}
```

Il gestore dell'evento può ricevere in ingresso un oggetto del DOM di tipo `event`, indicato con la notazione `$event`, che posso usare per accedere alle proprietà specifiche che varieranno a seconda del tipo di evento generato in uscita dal componente.

```
<button (click)="inviaDato($event)">Invia</button>
```

All'interno del metodo `inviaDato()`, posso sfruttare questo evento per accedere a diverse proprietà:

```
export class MioComponente {
  constructor() {
  }
```

(continua)

```

    inviaData(oggettoevento) {
        // qui posso usare l'oggetto event e le sue proprietà
        alert(oggettoevento.currentTarget.name);
    }
}

```

Qui sotto puoi vedere quelle più comuni legate all'evento click del mouse:

- *currentTarget* - Restituisce l'elemento su cui si è collegato il gestore di evento
- *target* - Restituisce l'elemento effettivo su cui si è verificato l'evento
- *target.name* - Restituisce il nome dell'elemento del DOM su cui si è verificato l'evento
- *target.getAttribute("class")* - Restituisce l'attributo class dell'elemento su cui si è verificato l'evento
- *target.getAttribute("id")* - Restituisce l'attributo id dell'elemento su cui si è verificato l'evento

8.2 Gestire il click su un elemento del componente <ca-listnews>

Vediamo come sia possibile sfruttare questi concetti, per recuperare delle informazioni dal click fatto su un elemento del componente <ca-listnews> progettato in precedenza.

Lo scopo è quello di poter recuperare l'identificativo associato a ciascun articolo, al fine poi di visualizzare il dettaglio di questo, prelevando ulteriori informazioni da un'ipotetica sorgente esterna.

Il template del componente, potresti modificarlo nel seguente modo:

```

<div class="news">
    <ul>
        <li *ngFor="let notizia of listanews" (click)="dettaglio(notizia.id)">
            <h2>{{notizia.titolo}}</h2>
            <p>{{notizia.descrizione}}</p>
        </li>
    </ul>
</div>

```

Come puoi osservare, ho aggiunto ad ogni tag , un evento “click” da monitorare, e un gestore d’evento, di nome *dettaglio()*, che riceve in ingresso l’identificativo *id* recuperato dall’oggetto notizia.

Non ho usato le doppie parentesi graffe per passare l' `id` della notizia al gestore di evento, perché ci troviamo a destra del simbolo di uguale.

Il componente, completo di gestore d'evento, potrebbe quindi diventare:

```
notizie/notizie.component.ts

import { Component } from '@angular/core';
import { News } from '../../model/news.model';

@Component({
  selector: 'ca-listanews',
  template: `
    <div class="news">
      <ul>
        <li *ngFor="let notizia of listanews" click)="dettaglio(notizia.id)">
          <h2>{{notizia.titolo}}</h2>
          <p>{{notizia.descrizione}}</p>
        </li>
      </ul>
    </div> `
})
export class NotizieComponent {
  listanews: News[] = [];
  constructor() {
    this.listanews = [{ id:1, titolo: 'Titolo Notizia 1', desc: 'Descrizione notizia 1' }];
  }
  // definisco il metodo dettaglio che riceve un id notizia
  dettaglio(id) {
    alert("Id Notizia: " + id);
  }
}
```

Non ho specificato il tipo di dato in ingresso alla funzione `dettaglio`, anche se è buona norma aggiungerlo, per rendere il codice maggiormente leggibile ed essere più "protetto" da eventuali errori.

Facendo il click su ogni notizia sia sopra al titolo che sopra alla descrizione, visualizzerò un messaggio di alert con la scritta: "Id Notizia: X" dove al posto di X sarà visualizzato il relativo id associato alla notizia cliccata.

8.3 Gestire gli eventi del mouse

Oltre al classico click, è possibile intercettare anche una serie di eventi legati al mouse. Qui di seguito i più usati, chiaramente in ambito di applicazioni realizzate per il web:

- *mouseup* - Emesso non appena si rilascia il pulsante del mouse dopo un click
- *mousedown* - Emesso non appena si schiaccia il pulsante del mouse sopra all'elemento
- *mousemove* - Emesso non appena si muove il mouse sopra all'elemento
- *mouseover* - Emesso non appena si posiziona il mouse sopra all'elemento o a uno dei suoi figli

Una lista completa di tutti gli eventi del DOM la puoi trovare a questo link: <https://developer.mozilla.org/en-US/docs/Web/Events>

Così come accade per l'evento "click" del mouse, anche per questi è possibile recuperare una serie di informazioni legate all'oggetto "evento", passato al gestore dell'evento, sotto forma di parametro dal nome \$event.

Tra questi, oltre a quelli visti in precedenza, posso recuperare:

- *clientX* - recupera la coordinata X dove è posizionato il mouse al momento del verificarsi dell'evento
- *clientY* - recupera la coordinata Y dove è posizionato il mouse al momento del verificarsi dell'evento

Ad esempio, riprendendo l'applicazione di domoticica creata nelle precedenti lezioni, potrei sfruttare l'azione del *mousedown*, per accendere o spegnere le lampadine presenti nel template, andando a modificare la proprietà `stato` di un oggetto Stanza.

Ora che sappiamo creare dei modelli di dati, potremmo aggiungere una struttura simile a questa, di nome `stanza.model.ts`, da importare nel mio componente:

model/stanza.model.ts

```
export class Stanza {
  stanza: string;
  stato: string;
  luminosita: string;
}
```

Riprendendo il codice del template *app-component.html* visto nelle precedenti lezioni, l'unica modifica da fare è l'inserimento di un ipotetico gestore di evento *onOff()* che andrà a monitorare il *mousedown*:

```
<div class="lucistanza">
  <ul>
    <li *ngFor="let lucisingole of lucidb"
        (mousedown)="onOff(lucisingole)">
      <span *ngIf="lucisingole.stato == 'OFF'; else showluce">
        <i class="material-icons">lightbulb_outline</i>
        {{lucisingole.stanza}}</span>
      <ng-template #showluce>
        <span><i class="material-icons on">lightbulb_outline</i>
        {{lucisingole.stanza}}</span>
      </ng-template>
    </li>
  </ul>
</div>
```

Al gestore d'evento, passo in ingresso proprio l'oggetto *lucisingole*, quindi nel corpo del metodo, andrà a modificare il valore della proprietà *stato*, a seconda che la luce sia nello stato di acceso “ON”, o nello stato di spento “OFF”.

```
onOff(lucestanza) {
  if(lucestanza.stato == 'OFF') {
    lucestanza.stato = 'ON';
  } else {
    lucestanza.stato = 'OFF';
  }
}
```

Ecco il codice completo del componente principale dell'app, che include la definizione del modello di dati:

```
app.component.ts

import { Component } from '@angular/core';
import { Stanza } from './model/stanza.model';

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

(continua)

```

export class AppComponent {
  show: boolean = true;
  lucidb: Stanza[];
  constructor() {
    this.lucidb = [
      {
        stanza: 'Luce Cucine',
        stato: 'OFF',
        luminosita: 5
      },
      {
        stanza: 'Luce Sala Pranzo',
        stato: 'ON',
        luminosita: 2
      },
      {
        stanza: 'Luce Bagno',
        stato: 'OFF',
        luminosita: 8
      }];
  }

  onOff(stanza) {
    if(stanza.stato == 'OFF') {
      stanza.stato = 'ON';
    } else {
      stanza.stato = 'OFF';
    }
  }
}

```

8.4 Gestire gli eventi della tastiera

L'oggetto `$event` visto in precedenza, diventa particolarmente utile non appena si debba gestire l'insieme degli eventi legati all'input di dati effettuato tramite i classici tag di un modulo web (`input`, `textarea`).

Parleremo in dettaglio di questi aspetti quando affronteremo la gestione dei moduli web, in quanto il recupero delle informazioni da questi campi, avviene in modo molto più semplice rispetto alle tecniche usate in JavaScript.

I tipici eventi che è possibile monitorare sono:

- `keydown` - si verifica non appena si preme un pulsante qualsiasi della tastiera
- `keyup` - si verifica non appena si rilascia un pulsante qualsiasi della tastiera
- `keyenter` - si verifica non appena si schiaccia il tasto `enter` della tastiera

Come sempre, dovrò sfruttare la notazione Angular per il collegamento di un evento, grazie alle parentesi tonde:

```
<taghtml (eventotastiera)="gestore()">..</taghtml>
```

Tra le informazioni che potresti avere la necessità di recuperare, c'è quella legata al tipo di tasto schiacciato. Basti pensare ad un gioco che dia la possibilità all'utente di usare i tasti "freccia" per spostare un oggetto nello schermo.

Così come visto per gli eventi collegati al mouse, sfruttando l'oggetto `$event` che sarà di tipo `KeyboardEvent`, sono in grado di recuperare queste informazione e non solo, come indicato qui sotto:

- `key` - il nome del tasto premuto
- `shiftKey` - valore booleano indicando se il tasto SHIFT (maiuscolo) è stato o meno premuto
- `ctrlKey` - valore booleano indicando se il tasto CTRL è stato o meno premuto
- `altKey` - valore booleano indicando se il tasto ALT è stato o meno premuto

Nell'applicazione seguente, ho inserito un tag input, collegato all'evento `keyup`, e un gestore d'evento a cui passerò l'oggetto `$event`, grazie al quale riuscirò a valorizzare un array che mi permetterà di visualizzare a video le diverse informazioni legate all'oggetto passato, memorizzate all'interno dell'array `eventiTastiera`:

```
tastiera/tastiera.component.ts

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

@Component({
  selector: 'ca-tastiera',
  template: `
    <h1>Eventi Tastiera</h1>
    <input (keyup)="checkTasto($event)">
    <table><thead></thead>
    <tbody>
      <tr *ngFor="let tastiera of eventiTastiera, let i=index">
        <td>{{i+1}}</td>
        <td>{{tastiera.key}}</td>
      </tr>
    </tbody>
  </table>
`)

export class TastieraComponent {
```

(continua)

```

eventiTastiera:Array<Event>;
constructor() {
    this.eventiTastiera = [];
}
checkTasto(event:KeyboardEvent) {
    this.eventiTastiera.push(event);
}
}

```

Se provassi a scrivere il mio nome Davide, otterrei:

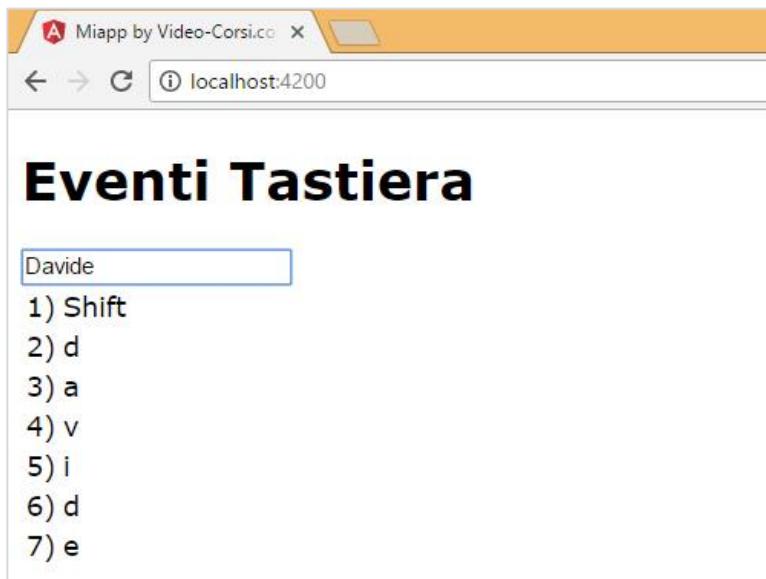


Figura 8.1 – Visualizzazione elenco tasti premuti della tastiera

Osserva come abbia usato una sintassi della direttiva `*ngFor` leggermente diversa da quella base vista in precedenza.

Grazie ad una seconda variabile `i`, valorizzata con la variabile locale `index` specifica di `*ngFor`, posso mostrare un conteggio del numero di volte in cui è ripetuto il ciclo.

Capitolo 9

Progettare Componenti “Intelligenti”

9.1 Cos’è un componente “intelligente”?

Fino ad ora abbiamo creato un'applicazione costituita da due soli componenti: quello predefinito, che si crea non appena creiamo un nuovo progetto e quello personalizzato, sviluppato per spiegare i diversi concetti del corso.

Abbiamo poi detto che per poter leggere il contenuto di una variabile definita nel template o per scriverci dei dati all'interno, devo sfruttare la tecnica del “Data Bindigs” e queste operazioni sono possibili solo all'interno dello stesso componente.

Nelle applicazioni Angular in realtà, avremo molti componenti e ognuno di questi avrà la necessità di **comunicare con gli altri**, sia ricevendo dei dati in ingresso (relazione genitore->figlio), che restituendo dei dati in uscita (relazione figlio->genitore).

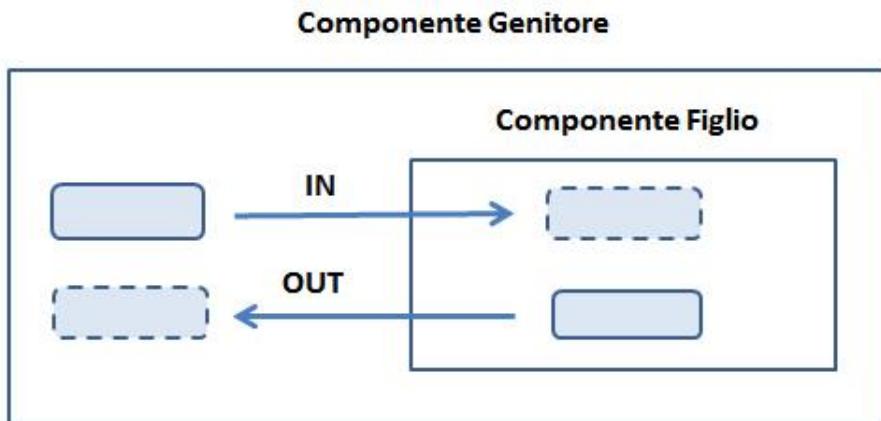


Figura 9.1 – Comunicazione bidirezionale tra componente genitore e figlio

Questa tipologia di progettazione, offre un ulteriore vantaggio, ossia la possibilità di riutilizzare alcuni componenti, in diverse sezioni dell'applicazione. Ad esempio, l'applicazione che mostrava l'elenco completo delle notizie, potrebbe essere dotata di altre sezioni, in grado di mostrare solo le notizie di una certa categoria.

In questo modo rendo il componente indipendente dai dati, che invece spetterà al genitore o padre, fornire.

In sostanza vado a **ridurre le responsabilità** del componente, allineandomi al principio detto "**SRP**" (**Single Responsibility Principle**), ossia che un componente deve assolvere solo uno specifico compito.

Quelli che inizieremo a progettare nelle prossime pagine, saranno dei componenti "intelligenti", ossia in grado di comunicare con altri componenti presenti nella vista che funge da contenitore.

In particolare ci concentreremo sulle tecniche per ricevere dei dati in ingresso proveniente dal padre, e sulle tecniche per comunicare a quest'ultimo il verificarsi di un opportuno evento.

In questo modo, sarà possibile riutilizzare il componente in più sezioni di una stessa applicazione. E' un concetto che ha qualche similitudine con quello delle classiche "funzioni" presenti in molti linguaggi di programmazione.

Ricordando l'obiettivo che ci siamo prefissati di realizzare per la prima schermata dell'applicazione, potremmo suddividerla in blocchi, come da *figura 9.2*.



Figura 9.2 – Visualizzazione dell'elenco treni in arrivo e suddivisione in blocchi

Si tratterà di definire una struttura ad albero per il template del componente *treni.component.ts*, che abbia all'interno un componente ex-novo *metro.component.ts*, rappresentativo del singolo treno, che chiaramente dovrà essere ripetuto in certo numero di volte a seconda di quanti treni sono in arrivo.

```
<h1>Treni in Arrivo</h1>
<div class="listatreni">
  <componente singolo treno>
  <componente singolo treno>
  ...
</div>
```

E' chiaro che questa struttura richiederà al componente genitore - *treni.component.ts* – di fornire a ogni componente figlio – *metro.component.ts* – i dati dei singoli treni, quindi dovremo imparare le tecniche per effettuare questo passaggio d'informazioni.

9.2 Proprietà d'ingresso a un componente con @Input

Uno dei meccanismi con cui è possibile passare dei dati da un **componente padre/genitore** a un componente **figlio**, è quello di sfruttare il "Property Binding" , ossia il collegamento di una proprietà presente nel template del padre, con una proprietà definita nel corpo della classe figlio, tramite il decoratore `@Input()` .

Riprendendo l'applicazione MetroChat, potrete mmo trasferire tutte le righe che permettevano la visualizzazione del singolo treno, all'interno di un componente dedicato, che diventerebbe così figlio del componente `<ca-treni>`.

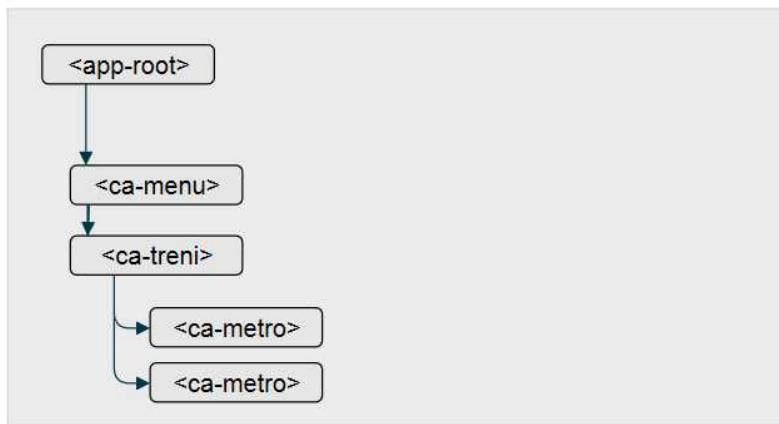


Figura 9.2 – Relazione genitore/figlio tra i vari componenti con riferimento alla visualizzazione dei treni in arrivo

Come selettore potremmo usare <ca-metro> e aggiungergli la direttiva *ngFor, in modo da ripeterlo a seconda di quanti treni sono stati memorizzati nell'array definito nel componente padre.

treni/treni.component.ts

```
import { Metro } from '../../model/metro.model';
import { Component, OnInit } from '@angular/core';
import { LISTAMETRO } from '../../dati/metroremoti';

@Component({
  selector: 'ca-treni',
  template:
    `<h1>Treni in Arrivo</h1>
    <div class="listatreni">
      <ca-metro *ngFor = "let metro of listametro" DATI-DA-INIETTARE>
        </ca-metro>
    </div>
  `
})

export class TreniComponent implements OnInit {
  listametro: Metro[];
  constructor() {
    this.listametro = [];
  }
  ngOnInit() {
    this.listametro = LISTAMETRO;
  }
}
```

Il componente con selettore <ca-metro>, viene ripetuto un certo numero di volte sulla base del numero di elementi presenti all'interno dell'array listametro, che per ora è stato valorizzato direttamente nel componente, con dei valori provvisori impostati con la costante LISTAMETRO.

Le operazioni d'inizializzazione, sono state inserite all'interno del metodo ngOnInit(), che sappiamo viene eseguito una sola volta, dopo l'esecuzione del costruttore.

Come facciamo a passare in ingresso al componente con selettore <ca-metro>, l'oggetto da visualizzare, che conterrà quindi tutti i dati?

Si usa la notazione con le doppie parentesi quadre:

```
<nome_selettore [proprietàIN] = "proprietà_padre"></nome_selettore>
```

Nel nostro caso, potremmo inserire una proprietà di nome `datiIn`, valorizzata con una proprietà del componente padre, ossia `metro`, che risulta essere una variabile definita nel template.

```
[datiIn] = "metro"
```

Si dice che `datiIn` è il “target”, ed è una proprietà che non esiste nel componente `<ca-treni>` ma solo nel componente `<ca-metro>`, mentre quello che c’è a destra dell’uguale è la sorgente, ed è una proprietà del componente padre `<ca-treni>`.

treni/treni.component.ts

```
import { Metro } from './model/metro.model';
import { Component, OnInit } from '@angular/core';
import { LISTAMETRO } from './dati/metroremoti';

@Component({
  selector: 'ca-treni',
  templateUrl: `
    <h1>Treni in Arrivo</h1>
    <div class="listatreni">
      <ca-metro *ngFor = "let metro of listametro" [datiIn]="metro">
        </ca-metro>
    </div>
  `
})
export class TreniComponent implements OnInit {
  listametro: Metro[];
  constructor() {
    this.listametro = [];
  }
  ngOnInit() {
    this.listametro = LISTAMETRO;
  }
}
```

Nel componente figlio, per contrassegnare la proprietà `datiIn` come d’ingresso e informare Angular che si tratta di una proprietà pubblica accessibile dall’interno della classe, si aggiunge il decoratore `@Input()`, a cui si fa seguire il nome di una proprietà che coincide con quella inserita all’interno del template del padre.

Tale decoratore, dovrà essere aggiunto nel corpo della classe figlio:

```
@Input() proprietàIN;
```

Pertanto il nuovo componente che creeremo sarà *metro.component.ts*, all'interno della cartella *metro*, e avrà come selettore proprio `<ca-metro>`. Inserendo solo alcuni dati essenziali, potrebbe essere scritto in questo modo:

metro/metro.component.ts

```
import { Component, Input, OnInit } from '@angular/core';
import { Metro } from '../../model/metro.model';

@Component({
  selector: 'ca-metro',
  template: `
    <p>Linea: {{datiIn.linea}}</p>
    <p>Partenza: {{datiIn.tempo}}</p>
  `
})

export class MetroComponent implements OnInit {
  // marchio la proprietà datiIn, come d'Ingresso
  @Input() datiIn: Metro;
  constructor() {
  }
  ngOnInit() {
  }
}
```

Come puoi notare, abbiamo inserito la riga:

```
@Input() datiIn: Metro;
```

che permette ad Angular di recuperare l'oggetto `Metro`, valorizzato ad ogni ciclo di esecuzione della direttiva `ngFor`, inserita nel componente genitore *treni.component.ts*

Così facendo, la variabile `datiIn` potrà essere usata, sia nel corpo della classe del componente con selettore `<ca-metro>`, sia nel template per accedere alle singole proprietà dell'oggetto.

Osserva inoltre come all'interno della lista dei moduli da importare da `@angular/core`, sia stato inserito `Input`, al fine di poter accedere al relativo decoratore.

Rammenta inoltre che il compilatore TypeScript, se non configurato come nel Capitolo 1, segnalerà un errore in quanto la variabile non è inizializzata in fase di compilazione. Puoi ovviare a questo inserendo l'operatore ! (*definite assignment assertion*) subito dopo il nome della proprietà:

```
@Input() datiIn!: Metro;
```

e verificare l'esistenza di un valore per la variabile sfruttando `ngOnInit`.

9.3 Definire più proprietà d'ingresso

Nel caso volessi inserire più proprietà, dovremmo aggiungere più blocchi con parentesi quadre. Nel template del componente padre potremmo scrivere:

```
<ca-metro [prop1]="valore1" [prop2]="valore2"></ca-metro>
```

mentre nella classe del componente figlio `<ca-metro>` dovremmo scrivere:

```
metro/metro.component.ts

export class MetroComponent implements OnInit {
  @Input() prop1;
  @Input() prop2;

  constructor() {
  }
}
```

In riferimento all'applicazione MetroChat, al fine di calcolare il tempo mancante alla partenza del treno, potremmo aggiungere un ulteriore parametro d'ingresso rappresentato dall'ora attuale. Ipotizzeremo che l'orario di partenza sia espresso in millisecondi a partire dalla data di Unix.

Pertanto se la proprietà `now` è valorizzata con:

```
this.now = new Date().getTime();
```

all'interno del template padre, dobbiamo scrivere:

```
<ca-metro ... [datiIn]="metro" [ora]="now"></ca-metro>
```

e nel componente `<ca-metro>`, aggiungeremo anche questa seconda proprietà di ingresso:

```
metro/metro.component.ts

import { Metro } from './model/metro.model';
import { Component, Input, OnInit } from '@angular/core';

@Component({
  selector: 'ca-metro',
  template: `<p>Linea: {{datiIn.linea}}</p>
              <p>Partenza: {{datiIn.tempo}}</p>
            `
})

export class MetroComponent implements OnInit {
  @Input() datiIn!: Metro;
  @Input() ora!: number;

  constructor() {}
  ngOnInit() {}
}
```

Sfrutteremo questa proprietà più avanti per simulare un conto alla rovescia.

9.4 Cambiare il nome alla proprietà di ingresso

La domanda che ora potresti farti è se all'interno delle parentesi tonde del decoratore `@Input()`, si può inserire qualcosa. In alcuni casi sì, quando ad esempio abbiamo la necessità di cambiare il nome associato alla proprietà d'ingresso.

Nell'esempio che segue, abbiamo inserito la stringa `nuovo_nome_prop1` che potrà essere usata al posto del nome originale `prop1`, definito nel componente genitore.

```
metro/metro.component.ts

export class MetroComponent implements OnInit {
  @Input('prop1') nuovo_nome_prop1;
  constructor() {}
}
// ...
```

9.5 Mostrare le informazioni su un componente di dettaglio

Ora che sappiamo trasmettere delle informazioni da un componente padre ad un componente figlio, potremmo aggiungere ulteriori funzionalità all'applicazione MetroChat.

In particolare, potremmo definire un componente con selettore `<ca-dettaglio>` all'interno della cartella `treni`, il cui scopo sarà quello di mostrare delle info rmazioni di dettaglio , non appena l'utente cliccherà su uno dei treni in arrivo.

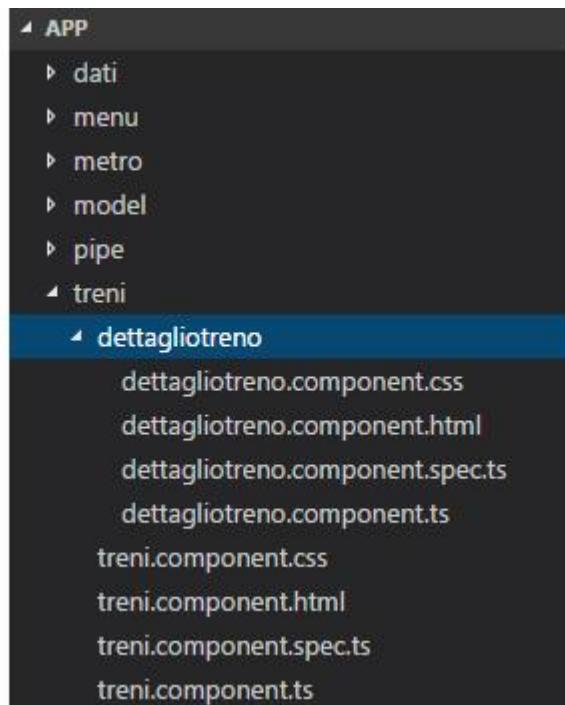


Figura 9.3 – Nuovo Componente per visualizzare le informazioni di dettaglio del treno

Lo andremo a inserire sempre nel template del componente `treni.component.ts`, proprio sotto alla lista dei treni rappresentata dai diversi componenti con selettore `<ca-metro>`

```
// ...
<ca-metro ...></ca-metro>
<ca-dettaglio [treno]="trenoselezionato"></ca-dettaglio>
// ...
```

La proprietà `trenoselezionato`, la dovremo valorizzare chiaramente con un opportuno gestore d'evento click fatto sul componente `<ca-metro>`, mentre `treno`, sarà la proprietà d'ingresso definita nel componente `<ca-dettaglio>` e che potrà leggere se contrassegnata con il decoratore `@Input()`.

```
<ca-metro *ngFor="let metro of listametro" [datiIn]="metro"
  (click)="setMetro(metro)">
</ca-metro>
```

Il metodo `setMetro()` non farà altro che valorizzare una proprietà della classe, con il riferimento all'oggetto `metro` appena visualizzato nella vista/template:

```
setMetro(t) {
  this.trenoselezionato = t;
}
```

Prima di sviluppare il codice del componente dettaglio, mettiamo insieme i diversi pezzi creati fino ad ora:

```
treni/treni.component.ts
```

```
import { Metro } from './model/metro.model';
import { Component, OnInit } from '@angular/core';
import { LISTAMETRO } from './dati/metroremoti';

@Component({
  selector: 'ca-treni',
  template: `
    <h1>Treni in Arrivo</h1>
    <div class="listatreni">
      <ca-metro *ngFor="let metro of listametro"
        [datiIn]="metro"
        [ora]="now"
        (click)="setMetro(metro)">
      </ca-metro>
    </div>
    <!-- aggiungo questo componente di dettaglio-->
    <ca-dettaglio [treno]="trenoselezionato"></ca-dettaglio>
  `
})
export class TreniComponent implements OnInit {
  listametro!: Metro[];
  trenoselezionato!: Metro;
  now!: number;
```

(continua)

```

constructor() {
    this.listametro = [];
}

ngOnInit() {
    this.listametro = LISTAMETRO;
}

setMetro(t) {
    this.trenoselezionato = t;
}
}

```

Il componente di dettaglio invece, mostrerà le diverse proprietà dell'oggetto ricevuto in ingresso. Per ora ci limitiamo a inserirne solo alcune, senza curare l'aspetto grafico:

treni/dettagliotreno/dettagliotreno.component.ts

```

import { Metro } from '../../../../../model/metro.model';
import { Component, Input } from '@angular/core';

@Component({
  selector: 'ca-dettaglio',
  template: `
<div *ngIf="treno">
  <h1>Dettaglio Treno</h1>
  <p>ID: {{treno.idt}}</p>
  <p>Linea: {{treno.linea}}</p>
  <button (click)="chiudi">Chiudi dettaglio</button>
</div>
`)

export class DettagliotrenoComponent {
  @Input() treno!: Metro;
  constructor() {
  }
}

```

Tale oggetto, in realtà, non è una copia dell'oggetto originario, ma è proprio lo stesso oggetto visualizzato nel componente padre. Ogni eventuale modifica alle sue proprietà, si rifletterà in automatico anche nella vista del componente padre.

I più attenti avranno notato che abbiamo aggiunto anche un tag `<div>`, con all'interno la direttiva `*ngIf`. Questo perché, il componente `<ca-dettaglio>` è

stato inserito nel template del componente `treni.component.ts`, che a sua volta fa parte integrante della vista che viene generata al caricamento dell'app.

Questa sezione deve essere visualizzata solo dopo che l'utente ha selezionato il treno d'interesse, quindi solo dopo aver valorizzato la variabile `trenoselezionato`, che viene poi collegata alla proprietà d'ingresso del componente figlio.

In assenza della direttiva `*ngIf`, serebbe segnalato un errore, perché tale proprietà è indefinita al caricamento del componente `<ca-app>` e di conseguenza `<ca-treni>`

Se invece inizializziamo la variabile `treno` nel costruttore del componente `treni.component.ts`, la sezione dettaglio sarà sempre visualizzata, anche in assenza di treno selezionato.

Esercitazione proposta

Con riferimento al componente `dettagliotreno.component.ts` appena progettato, realizzare il codice per il metodo “chiudi” al fine di far scomparire dalla visualizzazione i dati di dettaglio, non appena l'utente clicca sul pulsante “chiudi”.

9.6 Definire proprietà d'uscita con `@Output()`

Il passo che ci rimane da fare per creare un componente intelligente a 360 gradi, è capire come si possano intercettare anche dei dati in uscita.

In gergo si dice "iscriversi" alle proprietà d'uscita, con una terminologia che penso ti sia familiare se sai già cosa sono gli *Observable* (tratteremo a fondo questo concetto nel capitolo 15).

Noi sappiamo infatti che tutti i membri di un componente sono visibili solo all'interno di questo, quindi per fare in modo che un membro interno possa essere trasmesso anche al componente padre, dovrò "marchiarlo" in modo che Angular lo consideri affidabile e lo possa usare al proprio interno, sia nella classe che nel template.

In analogia a quello fatto per una proprietà d'ingresso, per definire una proprietà di uscita, la si "marchierà" con `@Output()`:

```
@Output() proprietàOut
```

Questa sintassi però non è completa, in quanto abbiamo detto che è necessario *iscriversi alla proprietà*, ossia trasformare il dato in uscita, in un evento da monitorare. Ad ogni emissione d'evento da parte del componente figlio, il corrispondente dato associato, verrà offerto al componente padre.

La sintassi completa da usare sarà:

```
@Output() proprietàOut = new EventEmitter<tipo>()
```

dove ho usato la classe `EventEmitter()`, che ci permette di creare degli eventi personalizzati. A questo punto `proprietàOut` è a tutti gli effetti un oggetto *evento*, in grado di **emettere degli eventi**, a cui posso associare dei valori in base al tipo specificato nell'EventEmitter.

A livello di componente padre, sappiamo già come si collega un gestore d'evento, quindi sfrutteremo la classica notazione delle parentesi tonde:

```
<tagHtml ... (proprietàOut)="azionePadre($event)"></taghtml>
```

Al posto del classico evento “click”, inseriremo il nome della proprietà in uscita definita nel componente figlio. Non appena `proprietàOut` emetterà l’evento con `emit`:

```
proprietàOut.emit(valore_emesso); // il figlio comunica al padre
```

questo verrà intercettato dal padre, che eseguire l’azione prevista (`azionePadre`), recuperando i dati emessi grazie all’oggetto `$event`. Applicando questi concetti ai due componenti `<app-root>` e `<ca-metro>`, e sfruttando il codice sviluppato per il pipe personalizzato, potrò scrivere:

```
import { Metro } from './model/metro.model';
import { Component, Input, Output, OnInit, EventEmitter }
        from '@angular/core';

@Component({
  selector: 'ca-metro',
  template: `
    <div [ngStyle]="stato">
      <p>Linea: {{datiIn.linea}}</p>
    </div>
  `
```

(continua)

```

<p>Partenza: {{datiIn.tempo}}</p>
<p>{{attesa | mmss}}</p>
<hr/>
</div>
})
export class MetroComponent implements OnInit {

    @Input() datiIn!: Metro;
    @Input() ora!: number;
    // marchio la proprietà in partenza, come d'Uscita
    @Output() inPartenza = new EventEmitter<string>();
    stato:Object;
    orapartenza: number;
    attesa:number;

    constructor() { }

    ngOnInit() {
        this.orapartenza = this.datiIn.tempo;
        // tempo mancante alla partenza
        this.attesa = this.orapartenza-this.ora;
        let x = setInterval(() => {
            this.attesa-=1000;
            if (this.attesa<=0) {
                // blocco il timer e mando l'evento in uscita
                clearInterval(x);
                // notifico il cambio di dato passando l'id del treno
                this.inPartenza.emit(this.datiIn.idt);
                // modiflico lo stato di visualizzazione del componente
                this.stato = {'display':'none'};
            }
        },1000);
    }
}

```

Come vedi, ho aggiunta la proprietà di uscita con nome `inPartenza`, assegnandogli l'istanza della classe `EventEmitter`, in modo da trasformarla in un oggetto in grado di emettere degli eventi con dei dati associati.

```
@Output() inPartenza = new EventEmitter<string>();
```

Il dato da emettere è l'identificativo del treno, che mi servirà per elencare, nel componente padre, tutti i treni partiti:

```
this.inPartenza.emit(this.datIn.idt);
```

Al fine di rendere il tutto più interessante, ho inserito questa riga all'interno di una funzione `setInterval()`, che decrementa di 1000 la proprietà `attesa` di ogni oggetto `Metro`, in modo da simulare il trascorrere del tempo.

Ti ricordo che la proprietà `tempo` è valorizzata con l'ora di partenza espressa in millisecondi. Questo il motivo per cui ho decrementato di 1000.

Ogni componente `<ca-metro>` avrà un conteggio separato dagli altri, e non appena si arriverà a 0, verrà bloccato il conteggio ed emesso il valore in uscita.

Nota come abbia usato il pipe creato in precedenza - `mmss.pipe.ts` - per formattare il tempo d'attesa, in minuti e secondi:

```
{{ attesa | mmss }}
```

Oltre a questo, ho aggiunto una direttiva `ngStyle` per cambiare la visibilità dell'elemento, impostando la proprietà CSS `display` al valore `none`.

Per poter intercettare il valore in uscita dal componente figlio `<ca-metro>`, dovrò inserire un gestore dell'evento `inpartenza`, direttamente nel template del componente padre:

```
...
<p>{{trenipartiti}}</p>
// mostro i treni partiti
<div class="listatreni">
  <ca-metro *ngFor="let metro of listametro"
    (inPartenza)="partiti($event)"
    [datiIn]="metro"
    [ora]="now"
    (click)="setMetro(metro)">
  </ca-metro>
</div>
<ca-dettaglio [treno]="trenoselezionato"></ca-dettaglio>
...
```

dove `inPartenza` è proprio la proprietà contrassegnata nel componente figlio e inserita come se fosse un evento personalizzato da monitorare. Il valore emesso, verrà gestito come dato in ingresso a 1 metodo `partiti()`, il quale si preoccupa

solo di valorizzare la proprietà `trenipartiti`, aggiungendo via via la stringa `id` di ogni treno.

```
partiti(id:string) {
    this.trenipartiti += " | " + id;
}
```

Il codice, completo delle varie inizializzazioni presenti nel costruttore, diventa:

```
treni/treni.component.ts

import { Metro } from './model/metro.model';
import { Component, OnInit } from '@angular/core';
import { LISTAMETRO } from './dati/metroremoti';

@Component({
  selector: 'ca-treni',
  template:
    <h1>Treni in Arrivo</h1>
    <div class="listatreni">
      <ca-metro *ngFor="let metro of listametro"
        (inPartenza)="partiti($event)"
        [datiIn]="metro"
        [ora]="now"
        (click)="setMetro(metro)">
      </ca-metro>
    </div>
    <p>{{trenipartiti}}</p>
    <!-- aggiungo questo componente di dettaglio-->
    <ca-dettaglio [treno]="trenoselezionato"></ca-dettaglio>
  `
})

export class TreniComponent implements OnInit {
  listametro: Metro[];
  trenoselezionato: Metro;
  trenipartiti:string;
  now:number;
  constructor() {
    this.trenipartiti = '';
    this.listametro = [];
    this.now = new Date().getTime();
  }
  ngOnInit() {
    this.listametro = LISTAMETRO;
  }
  setMetro(t) {
    this.trenoselezionato = t;
  }
}
```

(continua)

```

        partiti(id:string) {
            this.trenipartiti += " | " + id;
        }
    }
}

```

9.7 Accedere a membri di altri componenti con @ViewChild

In queste due ultime sezioni approfondiremo dei concetti che non useremo esplicitamente nell'applicazione MetroChat ma che è utile sapere per capire a fondo i principali meccanismi con cui è possibile far comunicare i componenti. Sen titi libero di saltarli per ora, se non vuoi mettere troppa carne al fuoco.

Abbiamo più volte detto che i membri interni ad un componente (proprietà e metodi) sono accessibili solo dall'interno di questo, a meno che non applichi le tecniche viste nei paragrafi precedenti.

Quindi un qualsiasi componente genitore, senza opportune tecniche, non potrà mai accedere a proprietà o metodi interni ad un componente figlio.

E' tuttavia possibile, con opportuni decoratori, accedere alla classe di un componente figlio, direttamente dal contenitore padre.

Si sfrutta il decoratore `@ViewChild`, con la seguente sintassi:

```
@ViewChild(ComponenteFiglio) proprietà: ComponenteFiglio;
```

Il decoratore `@ViewChild()` è una funzione che riceve in ingresso il nome del componente figlio e trova il corrispondente selettore presente nel template del componente padre (per l'esattezza il primo, nel caso ce ne siano più di uno e siano uguali).

In questo modo , nel componente padre possiamo usare un'istanza del componente figlio che ci permetterà di accedere ai metodi o alle proprietà interne alla classe.

Ad esempio, potresti creare un componente figlio di nome `popup.component.ts`, il cui scopo è quello di mostrare un'ipotetica scritta con un bottone per nasconderla.

Nel template del componente padre, potresti richiamarlo così:

```
<ca-popup>Messaggio in PopUp</ca-popup>
```

mentre il corpo del componente, potrebbe essere sviluppato in questo modo:

```
popup.component.ts

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

@Component({
  selector: 'ca-popup',
  template: `
    <div *ngIf="mostramsg">
      <ng-content></ng-content>
      <button (click)="nascondi()">OK</button>
    </div>
  `})
export class PopupComponent {
  mostramsg = false;

  mostra() {
    this.mostramsg = true;
  }

  nascondi() {
    this.mostramsg = false;
  }
}
```

Come puoi osservare il metodo `mostra()`, appartiene al componente `popup.component.ts`, e agisce sulla proprietà di nome `mostramsg` del componente stesso.

Potresti però avere la necessità di accedere a questo metodo dal componente contenitore - o padre -, al fine di far apparire la scritta , dopo il click su un elemento della vista del componente padre.

Per fare questo, nel componente genitore dovrai inserire il decoratore `@ViewChild()`, “iniettando” l’istanza del componente figlio all’interno di una proprietà.

Nell’ipotesi quest’ultima abbia nome `msg`, scriverò:

```
@ViewChild(PopupComponent) msg: PopupComponent;
```

NB: In Angular 8 era obbligatorio passare come secondo argomento anche l'oggetto `{static: false/true}`.

```
@ViewChild(PopupComponent, {static: false}) msg: PopupComponent;
```

Dalla versione 9 è stato di nuovo eliminato e il valore predefinito è `{static: false}`.

Il significato di questa proprietà è che se vogliamo che l'elemento referenziato sia disponibile durante l'aggancio al ciclo di vita `ngOnInit`, allora la proprietà `static` deve essere impostata a `true`.

Se invece possiamo attendere fino a dopo l'inizializzazione di tutta la vista, possiamo sfruttare il valore predefinito `false`, il che equivale ad essere disponibile in `ngAfterViewInit / ngAfterContentInit`.

Fatta questa precisazione, ora la proprietà `msg`, possiamo usarla per accedere direttamente al metodo `mostra()` del componente figlio:

```
this.msg.mostra();
```

I più attenti di voi, avranno notato nel template di `popup.component.ts` la presenza del tag `<ng-content>` non descritto in precedenza:

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

A cosa serve? Serve per visualizzare del contenuto. Quale? Quello inserito all'interno del tag html rappresentativo del componente che deve visualizzare il tale informazione.

```
<ca-popup>Messaggio in PopUp</ca-popup>
```

Nel nostro caso, sopra al bottone di “OK”, verrà visualizzata la scritta : “Messaggio in PopUp”, perché è in quel punto che ho inserito il tag `<ng-content>`.

Avrei potuto ometterlo e scrivere il testo direttamente all'interno del template del componente <ca-popup>, ma ho colto l'occasione al volo per introdurre anche questo concetto di Angular, utile per creare layout dinamici e capire quando entra in gioco il ciclo di vita ngAfterContentInit.

Il codice completo del componente padre *app.component.ts* diventa:

```

app.component.ts

import { Component, ViewChild } from '@angular/core';
import { PopupComponent } from './popup.component';

@Component({
  selector: 'app-root',
  template: `
    <ca-popup>Messaggio in popup</ca-popup>
    <button (click)="mostroMsg()">Mostro PopUp</button>
  `})
export class AppComponent {
  @ViewChild(PopupComponent) msg: PopupComponent;

  mostroMsg() {
    this.msg.mostra();
  }
}

```

Così come fatto per `@Input()` e `@Output()`, ricordati d'importare i riferimenti al decoratore `@ViewChild()` dalla libreria principale di Angular.

9.8 Accedere a membri di un componente figlio direttamente nel template

Nell'esempio precedente, siamo riusciti ad accedere a i membri di un componente figlio.

Angular offre un'ulteriore possibilità, ossia quella di usare una **variabile locale** nel template, che ci permette di accedere alla proprietà del componente a cui è collegata, sfruttando la notazione del cancelletto `#`. Viene chiamata in gergo **variabile template**.

```

<taghtml #nomevariabile></taghtml>

```

E' una tecnica che sfrutteremo anche quando parleremo di form. La differenza rispetto alla tecnica vista in precedenza, è che posso riferirmi a proprietà e metodi del componente figlio, solo nel template del componente padre e non nel corpo della classe.

Se all'interno del componente figlio <ca-popup> aggiungessi la variabile locale con nome #pp, per ottenere

```
<ca-popup #pp>Messaggio in popup</ca-popup>
```

potrei accedere nel template del componente padre ai metodi e alle proprietà definite internamente , con la notazione :

```
{ { pp.mostramsg } }           // accedo alla proprietà mostramsg
(click) = "pp.mostra()"      // accedo al metodo mostra
```

Pertanto potrei progettare il componente padre, in questo modo:

app.component.ts

```
import { Component } from '@angular/core';
import { PopupComponent } from './popup.component';

@Component({
  selector: 'app-root',
  template: `
    <ca-popup #pp>Messaggio in popup</ca-popup>
    <button (click)="pp.mostra()">Mostro PopUp</button>
    <p>Proprietà del componente figlio {{pp.mostramsg}}</p>
  `
}

export class AppComponent {
  mostroMsg() {
    this.msg.mostra();
  }
}
```

Tutte le volte che la proprietà mostramsg sarà modificata all'interno del componente figlio, automaticamente si aggiornerà il valore nel template del componente padre.

Questa tecnica può essere usata in combinazione con il decoratore `@ViewChild()` per accedere al componente sfruttando il nome della variabile template. Questo può essere utile nel caso avessimo più componenti e volessimo distinguere l'uno dall'altro.

```
app.component.ts

//...
@Component({
  selector: 'app-root',
  template: `
    <ca-popup #ppA>Messaggio in popupA</ca-popup>
    <ca-popup #ppB>Messaggio in popupB</ca-popup>
    <button (click)="mostroMsgA()">Mostro PopUpA</button>
    <button (click)="mostroMsgB()">Mostro PopUpB</button>
    <p>Proprietà del componente figlio {{pp.mostramsig}}</p>
  `})
export class AppComponent {
  @ViewChild('ppA') msgA: PopupComponent;
  @ViewChild('ppB') msgB: PopupComponent;

  mostroMsgA() {
    this.msgA.mostra();
  }

  mostroMsgB() {
    this.msgB.mostra();
  }
}
```

Esiste anche la possibilità di accedere a elementi nativi del DOM presenti nella vista, sebbene sia una tecnica sconsigliata perché si bypassa il sistema integrato di Angular che effettua i controlli per prevenire attacchi XSS. Questo sia che l'elemento appartenga al template del padre, sia a elementi HTML interni a qualche componente figlio.

Si sfrutta sempre una variabile template per referenziare il componente o l'elemento HTML e poi si fa riferimento alla classe `ElementRef` rappresentativa di un wrapper per ogni elemento nativo presente nel DOM e alla proprietà `nativeElement`.

```
app.component.ts

import { Component } from '@angular/core';
import { PopupComponent } from './popup.component';
@Component({
  selector: 'app-root',
  template: `
    <ca-popup #pp>Messaggio in popup</ca-popup>
    <button (click)="mostroMsg()" #miobtn>Mostro PopUp</button>
    <p>Proprietà del componente figlio {{pp.mostramsig}}</p>
  `)

```

(continua)

```
export class AppComponent {  
  
  @ViewChild(PopupComponent) msg: PopupComponent;  
  @ViewChild('miobtn') btn: ElementRef;  
  
  mostroMsg() {  
    // disabilito il bottone dopo il primo click  
    this.btn.nativeElement.disabled = true;  
    this.msg.mostra();  
  }  
}
```

In questo modo la variabile `btn` conterrà un riferimento al tag HTML `<button>` potendo così accedere a tutte le proprietà native. Chiaramente le stesse funzionalità si possono ottenere grazie alle tecniche viste di Data Binding, bypassando così i problemi di sicurezza.

Capitolo 10

La navigazione in Angular

10.1 Il concetto di Routing e Route

La gestione della transizione tra **diversi stati** di un'applicazione, è uno dei più importanti passi di progettazione che devi affrontare perché coinvolge sia un aggiornamento dell'url, che dello stato dell'applicazione.

La navigazione tra pagine di un sito web, è un concetto che penso tu conosca molto bene. Non appena inserisci un tag HTML di questo tipo:

```
<a href="paginaX.htm">Link alla pagina X</a>
```

il navigatore è in grado di prelevare dal server la paginaX e visualizzarla all'interno del browser. Il browser dal canto suo, mostrerà nella barra degli url, l'indirizzo completo della pagina, in relazione al dominio che si sta navigando, in modo che l'utente possa copiare l'indirizzo per memorizzarlo nei preferiti, inviarlo nei propri canali social, inserirlo in messaggi email etc.

Questo chiaramente nell'ipotesi il mio sito non sia una *Single Page Application (SPA)*, ossia una singola pagina.

Ogni pagina quindi è caratterizzata da un percorso fisico (URL) ben preciso. Tipici URL potrebbero essere:

```
www.miosito.com/
    /paginaX.htm
    /contattaci.php
    /catalogo/sedie.php
```

Nel caso di Angular sappiamo che il concetto di pagina non esiste, ma esiste il concetto di **vista o template**.

Essendo le viste tutte interne all'applicazione, non ci saranno chiamate al server, come avviene per un classico sito web. Questo permette un caricamento sicuramente più rapido di ogni sezione di un'applicazione o sito.

Tutti gli attuali browser supportano infatti la navigazione tra la cronologia degli url visitati dall'utente, questo grazie all'oggetto `history` e a dei metodi come `"history.pushState"`. Quest'ultimo ci permette di andare avanti e indietro sfruttando le classiche frecce di navigazione presenti in ogni browser. Possiamo così cambiare l'url (es. sfruttando JavaScript), senza inviare una richiesta al server.

In Angular, quindi, per implementare un sistema di navigazione, non è sufficiente inserire un link come si fa per le classiche pagine web, ma è necessario sudare un po' di più e sfruttare quello che si chiama in gergo tecnico **Router**.

Quest'ultima è una libreria opzionale, costituita da una combinazione di diversi "attori" che impareremo a conoscere via via nel corso delle prossime pagine.

Il meccanismo di navigazione tra le diverse *viste*, viene chiamato **Routing** e consiste nel predisporre un insieme di stringhe rappresentative del singolo indirizzo url a cui si vuole accedere, con associato un componente da visualizzare . Ognuna di queste stringhe viene chiamata **oggetto Route** o semplicemente `Route`.

- *Routing* > La navigazione in Angular intesa come passaggio da uno stato all'altro
- *Route* > Oggetto definito con due proprietà: una che identifica l'url e la seconda che identifica il componente con la vista da visualizzare

NB: Noi invece di usare la traduzione dall'inglese di *Route*, ossia "percorso, indirizzo", useremo semplicemente il termine `route`.

Pertanto il processo di configurazione o Routing in un'applicazione Angular, si effettua andando a configurare un `Router` con la lista delle `Route`, in modo che in corrispondenza alla navigazione ai diversi url definiti, il `Router` crei l'istanza del componente associato e mostri la relativa vista.

Questa configurazione è possibile solo se sfruttiamo un metodo presente nel modulo **RouterModule**, che dovremo quindi importare all'interno dell'`app.module.ts`

Il punto di partenza è lavorare all'interno del file `app.module.ts`, e aggiungere questo elemento.

10.2 Configurare il file AppModule

Abbiamo già detto in precedenza che il punto di partenza per l'introduzione di nuove funzionalità per un'applicazione Angular è il modulo radice, che per convenzione è chiamato *AppModule*.

Per gestire la navigazione, è necessario importare: 1) il modulo `RouterModule` dalla libreria `@angular/router`, che ci permetterà di sfruttare il metodo `RouterModule.forRoot()` per impostare l'array di route definite per l'applicazione e 2) `Routes` per permettere ad Angular di riconoscere l'array di oggetti che creeremo e che identificherà l'insieme delle diverse route.

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

All'interno dell'array della proprietà `imports` di `NgModule`, dovremo richiamare il metodo `forRoot()`, per definire un modulo, contenente il *Router* configurato:

```
import: [
  BrowserModule,
  RouterModule.forRoot(LISTA_ROUTES)
]
```

dove `LISTA_ROUTES`, vedremo dopo, sarà un array di tipo `Routes`, con all'interno le diverse route in grado di mostrare una particolare vista associata ad un componente.

Il codice completo dell'`AppModule` dell'applicazione MetroChat, pertanto diventerà:

```
app.module.ts

import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { RouterModule, Routes } from '@angular/router';
...
@NgModule({
  imports: [
    BrowserModule,
    RouterModule.forRoot(LISTA_ROUTES)
  ],
  declarations: [ MiaAppComponent, MenuComponent, ... ],
  ...
})
```

(continua)

```
bootstrap: [ MiaAppComponent ]
})
export class AppModule {}
```

Questa tecnica viene di solito usata quando non si hanno tante route da definire. Diversamente, al fine di non creare dei file AppModule troppo ricchi di codice, si dovrà creare un modulo separato da importare, cosa che per ora non faremo.

Il secondo passo da fare è definire le Routes.

10.3 Progettare l'array di route

L'applicazione vista fino ad ora prevedeva un componente <ca-menu> con definiti all'interno i diversi link per accedere alle ipotetiche pagine.

Nel caso dell'applicazione MetroChat, i link identificativi delle diverse viste potrebbero essere:

- www.miosito.com/
- www.miosito.com/preferiti
- www.miosito.com/login



Figura 10.1 – Menu applicazione MetroChat

Come facciamo a dire ad Angular quale vista mostrare e che indirizzo associargli?

Si definisce un array che conterrà la lista delle configurazioni per ogni route, dove ognuna di queste, sarà un oggetto literal JavaScript con un insieme di proprietà:

```
// singola route per mostrare la pagina di ingresso
{ path: '', component: TreniComponent }
```

Come vedi, abbiamo inserito due proprietà per l'oggetto JavaScript, le proprietà path e component. La prima identifica il percorso relativo al dominio principale che dovrà essere visualizzato nel browser, mentre la seconda, il nome scelto per

identificare la classe del componente che dovrà essere visualizzato in termini di vista o template.

Il nome della classe del componente NON deve comparire racchiuso tra virgolette e chiaramente deve esistere all'interno dell'applicazione.

Un controllo che ti invita a fare spesso è verificare che il componente associato alla route sia stato importato, ossia inserito all'interno dell'array della proprietà declarations di @NgModule().

Esistono diverse altre proprietà che è possibile specificare ma, per ora, limitiamoci a queste. La configurazione delle route per l'applicazione MetroChat, assumerà la forma di un array di tipo Routes:

```
[  
  { path: '', component: TreniComponent },  
  { path: 'preferiti', component: PreferitiComponent },  
  { path: 'login', component: LoginComponent }  
]
```

Abbiamo presupposto che l'applicazione sia costituita da due componenti aggiuntivi rispetto a quello principale, e i tre percorsi individuati, corrispondano proprio agli URL definiti all'inizio. Abbiamo creato il nostro primo array di route.

Questo array costituisce proprio l'argomento del metodo `forRoute()`, che abbiamo inserito nell'AppModule. Pertanto la successiva modifica da far e, sarà aggiungere tali righe all'interno del file AppModule:

```
app.module.ts

import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { RouterModule, Routes } from '@angular/router';
import { AppComponent } from './app.component';
import { MenuComponent } from './menu/menu.component';
import { TreniComponent } from './treni/treni.component';
import { LoginComponent } from './login/login.component';
import { PreferitiComponent } from './preferiti/preferiti.component';

@NgModule({
  imports: [
    BrowserModule,
    RouterModule.forRoot([
      { path: '', component: TreniComponent },
      { path: 'preferiti', component: PreferitiComponent },
      { path: 'login', component: LoginComponent }])
  ],
  (continua)
```

```

declarations: [ AppComponent, MenuComponent, TreniComponent,
PreferitiComponent, LoginComponent ],
bootstrap: [ AppComponent ]
}

export class AppModule {}

```

In questo modo, non appena l'utente inserirà nell'url un link corrispondente a quelli visti all'inizio, Angular troverà una corrispondenza con i percorsi indicati nel file e sarà in grado di visualizzare la vista del relativo componente.

La domanda che potresti farti è: ma dove verrà visualizzato il template del componente? Un componente, abbiamo sempre detto, è solo un piccolo pezzo dell'interfaccia di un'applicazione.

Quest'ultima avrà sempre un componente principale, avviato al caricamento dell'app, tramite il file *index.html*.

Con riferimento alla nostra applicazione, il primo componente caricato è *app.component.ts*, che ha questo template:

```

//...
@Component({
  selector: 'app-root',
  template: `
    <ca-menu><ca-menu>
      <h1>MetroChat</h1>
    </ca-menu>
  `})
//...

```

Fino ad ora, sotto la scritta MetroChat, abbiamo aggiunto all'occorrenza il selettore del template in fase di progettazione, considerato che per ora abbiamo sviluppato solo una sezione dell'applicazione.

Nel caso di più sezioni, dovremo trovare un sistema in grado di far visualizzare sotto al menu, gli ulteriori componenti che intendiamo realizzare, come il dettaglio, i preferiti o il login.

10.4 Indicare dove visualizzare le route con RouterOutlet

Per indicare ad Angular, dove visualizzare il componente associato ad una particolare route, ci viene in aiuto una direttiva fornita dal modulo *RouterModule*.

Quest'ultimo è già stato inserito in precedenza nel file AppModule quindi è disponibile per ogni componente dell'applicazione.

La direttiva si chiama `routerOutlet` e si inserisce direttamente nel template del componente radice, come una sorta di "segnaposto" per tutta l'applicazione:

```
<ca-menu><ca-menu>
<h1>MetroChat</h1>
<router-outlet></router-outlet>
```

Ora Angular sa prà che sotto al selettore `<router-outlet>`, dovrà aggiungere il template del componente che corrisponde all'indirizzo di volta in volta presente nella barra degli indirizzi del browser.

Nel caso dell'applicazione MetroChat, in funzione delle route già create in precedenza, avremo queste corrispondenze:

1) Url: www.miosito.com - Corrispondenza con il path vuoto "", quindi mostro TreniComponent

```
<ca-menu><ca-menu>
<h1>MetroChat</h1>
<router-outlet></router-outlet>
<!-- qui sotto Angular mostrerà il template di TreniComponent--&gt;</pre>

```

2) Url: www.miosito.com/preferiti - Corrispondenza con il path 'preferiti', quindi mostro PreferitiComponent

```
<ca-menu><ca-menu>
<h1>MetroChat</h1>
<router-outlet></router-outlet>
<!-- qui sotto Angular mostrerà il template di PreferitiComponent--&gt;</pre>

```

3) Url: www.miosito.com/login - Corrispondenza con il path 'login', quindi mostro LoginComponent

```
<ca-menu><ca-menu>
<h1>MetroChat</h1>
<router-outlet></router-outlet>
<!-- qui sotto Angular mostrerà il template di LoginComponent--&gt;</pre>

```

Ora, presi dall'entusiasmo, potremmo testare l'applicazione e cliccare sui diversi menu del componente <ca-menu> sviluppato nel capitolo 3:

```
menu/menu.component.html
```

```
<nav>
  <li><a href="">In arrivo</a></li>
  <li><a href="/preferiti">Preferiti</a></li>
  <li><a href="/login">Login</a></li>
</nav>
```

Ahimè non accadrebbe nulla, questo perché in Angular, per formare un link in grado di comunicare con il Router è necessario utilizzare una direttiva, sempre fornita da RouterModule. E' la direttiva RouterLink.

Come puoi constatare, le cose sono leggermente più complesse di quelle che sei abituato a fare con le classiche pagine web.

A questo punto, ecco altre domande che potresti farti:

- Come faccio a creare dei link per accedere alle diverse route?
- Posso aggiungere più <router-outlet> nel template?
- Cosa succede se non viene trovata alcuna corrispondenza tra URL e route definite?
- Come posso reindirizzare il navigatore ad un'altra vista/pagina?
- Come posso aggiungere dei parametri query all'URL?

Tutte cose a cui risponderemo nelle prossime pagine.

10.5 Configurare i link nel template con RouterLink

Per spostarci tra le diverse route, abbiamo bisogno di un menu. Questo menu è già stato progettato nella forma base, ma abbiamo visto che non funziona, nel senso che non ci permette di spostarci su una nuova pagina o vista.

Vediamo allora che modifiche apportare, al fine di farlo funzionare. Abbiamo detto che è necessario sfruttare la direttiva RouterLink.

Ecco la sintassi base:

```
<a routerLink="PERCORSO">Link alla Route</a>
```

Quando parleremo di link dinamici, vedremo come sfruttare un array al posto di PERCORSO. Come puoi osservare, è stato sostituito al classico `href`, il nome della direttiva.

Pertanto il template del componente con selettore `<ca-menu>`, potrebbe diventare:

```
menu/menu.component.html
```

```
<nav>
  <li><a routerLink="">In arrivo</a></li>
  <li><a routerLink="/preferiti">Preferiti</a></li>
  <li><a routerLink="/login">Login</a></li>
</nav>
```

Ricordati che questi url sono finti, nel senso che non fanno riferimento a pagine reali sul server, ma a percorsi relativi alla stessa applicazione. Questo è il motivo per cui nel file `index.html` è presente il tag HTML `<base href="/">`

Avendo anteposto ad ogni url il simbolo `/`, sono sicuro che il percorso fa sempre riferimento alla radice dell'app.

Se invece il link si trova all'interno di un componente visualizzato accedendo ad una route diversa dalla radice dell'app, varranno sempre le regole:

- “`/`” faccio sempre riferimento alla radice dell'app
- “`./`” o nullo, faccio riferimento allo stesso livello del componente
- “`../`” salgo di un livello rispetto alla posizione attuale

Fortunatamente ce la siamo cavata con una modifica di poco conto, andando a sostituire a `href`, la nuova direttiva. Nella realtà, quest'ultima offre diverse funzionalità che impareremo a sfruttare meglio quando parleremo di parametri dinamici o URL con query.

A questo punto se provassi a cliccare su i diversi link, finalmente riusciremmo a visualizzare le diverse "pagine" o sezioni dell'applicazione.

10.6 Progettare Template indipendenti dal componente radice

Una domanda legittima che potresti farti è: nel caso volessi progettare un'applicazione in cui non tutte le pagine devono mostrare il menu, come posso fare?

Abbiamo detto che tutte le applicazioni Angular hanno una struttura ad albero, dove tutti i componenti interni sono figli del componente radice `app.component.ts`, che è

il primo che viene caricato . Il template di quest'ultimo era stato progettato in questo modo:

```
// ...
<ca-menu><ca-menu>
<h1>MetroChat</h1>
<router-outlet></router-outlet>
// ...
```

Così facendo, tutte le route che di volta in volta configuriamo, saranno associate a dei componenti, i cui template saranno visualizzati sotto la direttiva `<router-outlet>`. Questo implica che in ogni pagina ci sarà sia il menu, che la scritta MetroChat.

Per ovviare a questo comportamento, la soluzione è molto semplice , perché sarà sufficiente eliminare tutti i tag dal componente radice e lasciare solo la direttiva `<router-outlet>`.

In questo modo, ogni componente associato alle diverse route, sarà visualizzato senza venire "sporcato" dagli elementi presenti nel template del c omponente radice, potendo così decidere se inserire in ogni template, un menu di navigazione oppure no.

Per l'applicazione MetroChat, possiamo dunque riprogettare tutti i template dei singoli componenti creati in precedenza, aggiungendogli il componente `<ca-menu>`.

1) Template **componente radice AppModule (<app-root>)**

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

2) Template **componente TreniComponent (<ca-treni>)** con all'interno altri due componenti

```
<ca-menu></ca-menu>
<h1>Treni in arrivo</h1>
<!--lista componenti metro -->
<ca-metro *ngFor="..." ...></ca-metro>
<ca-dettaglio ...></ca-dettaglio>
```

3) Template **componente PreferitiComponent (<ca-preferiti>)**

```
<ca-menu></ca-menu>
<h1>Preferiti</h1>
<!--lista preferiti-->
<ca-chat *ngFor="..." ...></ca-chat>
```

4) Template **componente LoginComponent (<ca-login>)**

```
<ca-menu></ca-menu>
<h1>Login App</h1>
<form>
  <label>Username</label>
  <input type="text" name="username">
  <label>Password</label>
  <input type="password" name="password">
  <button type="submit">Login</button>
</form>
```

10.7 Progettare Route di Redirect

L'ipotesi che sta alla base delle route progettate fino ad ora, è che l'utente acceda alle diverse sezioni dell'applicazione, usando i link del menu o accedendo direttamente dall'URL.

Una delle operazioni che in alcuni casi si ha la necessità di fare è reindirizzare il navigatore, che accede ad un particolare URL, a una pagina specifica in seguito ad un particolare evento.

Tipico caso è quello legato ad un redirect non appena l'utente apre l'applicazione, quindi accede all'URL predefinita, oppure dopo che si è aggiornato un dato su un database remoto.

Nel primo caso è necessario creare una corrispondenza tra l' URL predefinito associato al caricamento dell'applicazione, e la path della route vuota, al fine di reindirizzare il navigatore ad una path diversa.

La sintassi da usare per questo tipo di route, sarà:

```
{ path: '', redirectTo: '/NUOVOURL', pathMatch: 'full' }
```

dove NUOVOURL, è il nome di una stringa rappresentativa dell' URL associato a una route già creata in precedenza.

Inoltre sono state aggiunte nuove proprietà all'oggetto Routes, tra cui `redirectTo`, che ci permette di specificare a quale URL indirizzare il navigatore e `pathMatch`, per indicare che il confronto deve essere fatto su tutto l'URL e non su sottosezioni di questo.

Non appena l'utente avvia l'applicazione, quindi non appena viene caricato l' URL predefinito, Angular trova la corrispondenza con la path vuota " " e viene fatto un indirizzamento a una nuova route, che condurrà il navigatore al componente finale.

Nel caso dell'applicazione MetroChat, potremmo reindirizzare l'utente alla route con `path` impostato alla stringa "inarrivo" che mostrerà la lista dei treni in arrivo, quindi il componente `TreniComponent`.

Il file di configurazione delle route dell'applicazione, pertanto diventerà:

```
[  
  { path: 'inarrivo', component: TreniComponent },  
  { path: 'preferiti', component: PreferitiComponent },  
  { path: '', redirectTo: '/inarrivo', pathMatch: 'full' }  
]
```

mentre il template del menu di navigazione, cambierà solo in corrispondenza al primo link:

```
<nav>  
  <li><a routerLink="/inarrivo">In arrivo</a></li>  
  <li><a routerLink="/preferiti">Preferiti</a></li>  
  <li><a routerLink="/login">Login</a></li>  
</nav>
```

Nel caso, invece, volessimo ottenere un re-indirizzamento, non sulla base di un URL specifico, ma sulla base del verificarsi di un particolare evento, dobbiamo sfruttare il metodo `navigate()` di Router, che dovrà essere iniettato nel costruttore con un meccanismo che impareremo in dettaglio quando parleremo di "Services".

La classe Router dovrà essere importata dal package `@angular/router`:

```
import {Router} from '@angular/router';
```

La sintassi, invece, da usare nel corpo della classe del componente, sarà:

```
this.router.navigate(['PERCORSO']);
```

dove `router` è il nome che abbiamo scelto per l'oggetto iniettato nel costruttore della classe.

```
// ...
class MioComponente {
  constructor(private router: Router) {
  }
  reindirizzoHome() {
    this.router.navigate(['/']);
  }
}
// ...
```

Il parametro impostato per il metodo `navigate()`, è un array con specificato il percorso/path a cui reindirizzare l'utente. Nell'esempio indicato sopra, reindirizzeremo ad una pagina precedente già visitata e memorizzata nella cronologia del browser.

Chiaramente questo è l'uso base del metodo, ma più avanti vedremo come passare anche dei parametri query nell'URL.

10.8 Progettare Route di Pagina non trovata

Nel caso l'utente inserisca nell' URL un percorso non configurato nell'array di path, dobbiamo prevedere un percorso alternativo che rimandi il navigatore ad una classica pagina del tipo: "Errore 404: pagina non trovata"

Ipotizzando di aver creato il componente `Error404.component.ts`, in questo modo:

```
error404/error404.component.ts

import { Component } from '@angular/core';
import { Router } from '@angular/router';

@Component({
  template: `
    <h1>Pagina Non Trovata</h1>
    <button (click)="backToHome()">Torna alla Home</button>
  `
})
```

(continua)

```
class Error404Component {
  constructor(private router:Router) {
  }
  backToHome() {
    this.router.navigate(['/']);
  }
}
```

La sintassi da usare per configurare la route, sarà:

```
{ path: '**', component: Error404Component }
```

Al posto della classica stringa identificativa del percorso, ho aggiunto due simboli di asterisco, che stanno ad indicare la corrispondenza con una qualsiasi route.

NB: E' importante assicurarsi che questa sia l'ultima delle righe di configurazione delle diverse path, altrimenti l'applicazione visualizzerà sempre e solo il template del componente Error404Component.

In alternativa, potremmo usare un re-indirizzamento verso un percorso specifico (es. errore), a cui sarà associata la visualizzazione del componente Error404Component:

```
{ path: '**', redirectTo: '/errore', pathMatch: 'full' }
```

10.9 Progettare Route con figli

L' applicazione MetroChat, prevede un menu di navigazione ad un solo livello di profondità, dove con profondità , intendiamo l'equivalente numero di sottocartelle di un classico sito web.

Potrebbe però capitare di avere la necessità di creare degli URL con più livelli, come nel caso qui sotto:

/inarrivo	1° livello di profondità (padre)
/inarrivo/dettaglio	2° livello di profondità (figlio)
/inarrivo/dettaglio/ <i>ID</i>	3° livello di profondità (figlio)

La path con l'aggiunta del dettaglio, è una sottosezione dell'URL *inarrivo*, così come quella con l'aggiunta del dettaglio e del parametro *ID*, il quale potrebbe essere un parametro dinamico rappresentativo di una query d'interrogazione.

Per progettare Routes come queste e fare in modo che Angular trovi la corrispondenza tra l' URL inserito e la path configurata, si utilizza una particolare sintassi, che fa uso della proprietà `children` di `Routes`, andando a inserire all'interno di un array, i percorsi dei componenti figlio da visualizzare.

```
{ path: 'STRINGA', component: 'COMPONENTE_PADRE', children: [ALTRI_PATH] }
```

Il file di configurazione pertanto diventa:

```
{ path: 'inarrivo', component: TreniComponent},
{ path: 'inarrivo/dettaglio', component:
  DettagliotrenoComponent, children: [
    { 'path': '', redirectTo: 'error', pathMatch: 'full' },
    { 'path': 'rosso', component: TrenorossoComponent },
    { 'path': 'giallo', component: TrenogialloComponent }
  ],
  { path: 'preferiti', component: PreferitiComponent },
  { path: 'login', component: LoginComponent },
  { path: '', redirectTo: '/inarrivo', pathMatch: 'full' },
  { path: 'error', component: Error404Component },
  { path: '**', redirectTo:'error', pathMatch: 'full' }
```

La path *"/inarrivo"* rappresenta la route padre, mentre *"inarrivo/dettaglio"* una seconda route, associata al componente *dettagliotreno.component.ts*.

La ricerca della corrispondenza tra l'URL e le route, avviene in sequenza, andando a vedere prima se c'è la route con path *"inarrivo/dettaglio"* e se soddisfatta, proseguendo con l'analisi dei figli.

All'interno della proprietà `children`, abbiamo inserito un array di route, con la stessa struttura di una route classica, con la differenza che all'interno di `path`, abbiamo aggiunto solo la parte di URL che segue la stringa *"inarrivo/dettaglio"*.

In assenza di altri dati nell'URL oltre a *"inarrivo/dettaglio"*, reindirizzo il navigatore a un URL con path *"error"* che mostrerà il componente associato alla pagina non trovata.

In presenza di un'ulteriore stringa nell'URL - "giallo" o "rosso" - aggiungerò alla visualizzazione del componente `DettagliotrenoComponent`, anche il componente associato a queste due route.

In quale punto? Beh all'interno del selettore `<router-outlet>` presente nel template del componente `dettagliotreno.component.ts`.

Quando parleremo di route con parametri o route dinamiche, vedremo come sostituire alle due ipotetiche stringhe "giallo" e "rosso", una stringa generica, che ci permetterà di creare una corrispondenza tra diversi URL e un singolo componente da visualizzare nel `<router-outlet>`.

Questo processo lo posso ripetere per tutte le route figlio che intendo associare a particolari percorsi URL.

In particolare abbiamo inserito anche la route vuota "", al fine di reindirizzare il navigatore ad una pagina di errore evitando così che l'utente veda una pagina senza dei dettagli.

In questo modo possiamo gestire un maggior numero di URL e creare così ulteriori sezioni e componenti specifici per l'applicazione:

- "/" - componente `TreniComponent`
- "/inarrivo" - componente `TreniComponent`
- "/inarrivo/dettaglio" - componente `Error404Component`
- "/inarrivo/dettaglio/giallo" - componente `TrenorossoComponent`
- "/inarrivo/dettaglio/rosso" - componente `TrenogialloComponent`
- "/preferiti" - componente `PreferitiComponent`
- "/login" - componente `LoginComponent`
- "/error" - componente `Error404Component`
- "/urlgenerica" - componente `Error404Component`

10.10 Impostare il Template della route di dettaglio per l'applicazione MetroChat

Una delle modifiche più interessanti che potremmo fare all'applicazione, riguarda i dati di dettaglio di un treno, che fino ad ora, venivano visualizzati nel template del componente con selettore `<ca-treni>`.

Ad ogni treno si potrebbe associare un identificativo alfanumerico, da passare nell'URL come query, o non appena si digita un URL simile a questo:

/inarrivo/dettaglio/XXX

dove al posto di XXX , sostituiremo di volt a in volta, una stringa corrispondente all'identificativo del treno.

Ancora non conosciamo le tecniche per creare delle route con parametri dinamici, quindi per testare il funzionamento della route con children, abbiamo inserito manualmente questo valore nel link del componente di dettaglio, usando due ipotetiche stringhe (“rosso” e “giallo”) già codificate all’interno delle route create in precedenza.

Il template del componente genitore *dettagliotreno.component.ts*. diventerà:

```
<ca-menu></ca-menu>
<h1>Dettaglio Treno</h1>
<a routerLink='giallo'>Treno Giallo</a>-
<a routerLink='rosso'>Treno Rosso</a>
<router-outlet></router-outlet>
```

In questo modo potremo visualizzare i dati di dettaglio di ogni treno, sotto al selettore *<router-outlet>*.

Per quanto riguarda invece i template dei componenti associati alle route figlio, ad esempio il componente *trenorosso.component.ts*, non avrà particolari informazioni tranne quelle legate ai dettaglio del treno, ad esempio:

```
<p>ID Treno: Rosso</p>
```

Ricapitolando, il file *app.module.ts* con i nuovi componenti, diventa:

```
app.module.ts

import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { RouterModule, Routes } from '@angular/router';
import { AppComponent } from './app.component';
import { MenuComponent } from './menu/menu.component';
import { TreniComponent } from './treni/treni.component';
import { DettagliotrenoComponent } from
'./treni/dettagliotreno/dettagliotreno.component';
import { TrenorossoComponent } from
'./treni/dettagliotreno/trenorosso/trenorosso.component'; (continua)
```

```

import { TrenogialloComponent } from
'./treni/dettagliotreno/trenogiallo/trenogiallo.component';
import { PreferitiComponent } from
'./preferiti/preferiti.component';
import { LoginComponent } from './login/login.component';
import { Error404Component } from './error404/error404.component';

@NgModule({
  imports: [
    BrowserModule,
    RouterModule.forRoot([
      { path: 'inarrivo', component: TreniComponent },
      { path: 'inarrivo/dettaglio', component: DettagliotrenoComponent,
        children: [
          { path: '', redirectTo: '/error', pathMatch: 'full' },
          { path: 'rosso', component: TrenorossoComponent },
          { path: 'giallo', component: TrenogialloComponent },
        ]
      },
      { path: 'preferiti', component: PreferitiComponent },
      { path: 'login', component: LoginComponent },
      { path: '', redirectTo: '/inarrivo', pathMatch: 'full' },
      { path: 'error', component: Error404Component },
      { path: '**', redirectTo: 'error', pathMatch: 'full' }
    ])
  ],
  declarations: [
    AppComponent,
    MenuComponent,
    TreniComponent,
    DettaglioTrenoComponent,
    PreferitiComponent,
    LoginComponent,
    TrenorossoComponent,
    TrenogialloComponent
  ],
  bootstrap: [ AppComponent ]
})
export class AppModule {}

```

in cui, oltre alla definizione della nuova route, ho importato anche i nuovi componenti **DettagliotrenoComponent**, **TrenorossoComponent**, **TrenogialloComponent** aggiungendoli all'array della proprietà declarations.

Questi ultimi due, saranno sostituiti da un unico componente dettaglio, non appena impareremo a creare route con parametri dinamici.

Il componente *dettagliotreno.component.ts*, sarà:

treni/dettagliotreno/dettagliotreno.component.ts

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

@Component({
  selector: 'ca-dettagliotreno',
  template: `<ca-menu></ca-menu>
    <h1>Dettaglio Treno</h1>
    <a routerLink='giallo'>Treno Giallo</a>-
    <a routerLink='rosso'>Treno Rosso</a>
    <router-outlet></router-outlet>` 
})

export class DettagliotrenoComponent {
  constructor() {} 
}
```

mentre i due componenti provvisori *trenorosso.component.ts* e *trenogiallo.component.ts*, avranno una struttura simile:

treni/dettagliotreno/trenorosso/trenorosso.component.ts

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

@Component({
  selector: 'ca-trenorosso',
  template: `<p>Treno Rosso</p>` 
})

export class TrenorossoComponent {
  constructor() {} 
}
```

Ora che abbiamo capito come si creano URL che possano essere intercettati con opportune route figlio, vediamo di applicare queste tecniche per visualizzare su una nuova "pagina", il dettaglio di ogni treno, non appena l'utente clicca sul componente `<ca-metro>`.

Non conoscendo a priori quale sarà l'identificativo associato a ciascun treno, dovremo imparare la tecnica per creare una route con parametri dinamici al fine di recuperarli dall'URL e usarli per recuperare i dati del treno.

10.11 Progettare Route con parametri dinamici

Ora che abbiamo capito il meccanismo con cui creare delle route con figli, sfruttiamolo per realizzare la sezione dettaglio dell'applicazione MetroChat.

Abbiamo detto che questa sezione dovrà essere visualizzata in modo separato dalla lista dei treni, in modo che l'utente, una volta cliccato sul treno, acceda ad una nuova sezione dell'applicazione che mostri le ulteriori informazioni associate al treno.

Inoltre dovremo modificare il menu, in modo che venga visualizzato solo un link per tornare alla schermata principale.

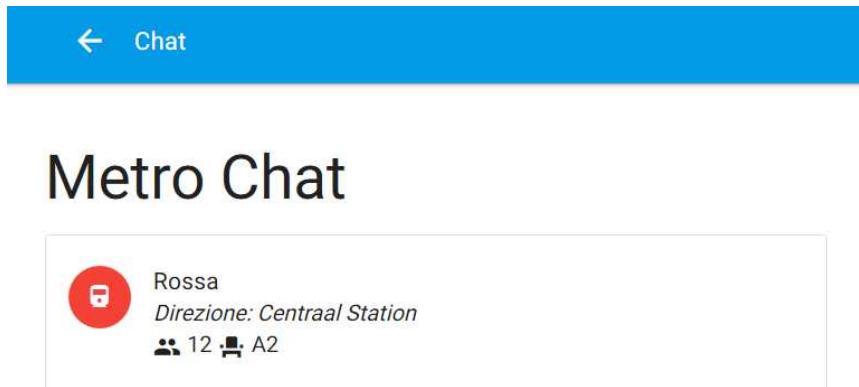


Figura 10.2 – Sezione di dettaglio dell'applicazione MetroChat

E' chiaro che il primo passo da fare è proprio di rogettare le route in modo che Angular sia in grado di trovare la corrispondenza con URL dinamici del tipo:

*inarrivo/dettaglio/*ID**

dove con ID, intendo una qualsiasi stringa rappresentativa dell'identificativo del treno.

Pertanto URL validi potrebbero essere:

inarrivo/dettaglio/AFD

inarrivo/dettaglio/FDE

etc.

E chiaro che, a priori, non posso creare decine di route associate ai possibili ID, perché in genere gli ID sono molteplici e nel tempo possono essercene di nuovi.

La soluzione è conoscere la tecnica per creare una sorta di “segnaposto” nella definizione della route.

La sintassi che si usa è questa:

```
{ path: 'percorso/:id', component: COMPONENTE }
```

Come vedi ho inserito il parametro :id al posto di una stringa generica.

Questa notazione permette ad Angular di capire che al posto di :id, dovrà aspettarsi una serie di parametri variabili, che potranno essere stringhe o numeri.

NB: E' importante specificare i due punti che devono essere attaccati al nome scelto per identificare il segnaposto del parametro dinamico. Quest'ultimo può essere indicato con una qualsiasi stringa valida, non necessariamente con id.

Tornando allora alla lista di route configurate in precedenza, al posto dei componenti di prova associati alle stringhe "rosso" e "giallo", potremmo aggiungere un nuovo componente generico *dettaglio.component.ts*, che sarà associato ad una path con un parametro dinamico.

```
{ path: 'inarrivo', component: TreniComponent },
{ path: 'inarrivo/dettaglio', component: DettagliotrenoComponent,
  children: [
    { path: '', redirectTo: 'error', pathMatch: 'full' },
    { path: 'giallo', component: TrenogialloComponent },
    { path: 'rosso', component: TrenorossoComponent },
    { path: ':id', component: DettaglioComponent }
  ]
},
{ path: 'preferiti', component: PreferitiComponent },
{ path: 'login', component: LoginComponent },
{ path: '', redirectTo: '/inarrivo', pathMatch: 'full' },
{ path: 'error', component: Error404Component },
{ path: '**', redirectTo:'error', pathMatch: 'full' }
```

NB: Fai attenzione che in questo caso l'ordine di apparizione delle diverse path, ha una sua importanza. Difatti, se inserissi la path dinamica, prima della path associata al componente Giallo o Rosso, questi ultimi non verrebbero mai visualizzati: Angular troverebbe subito una corrispondenza con la path dinamica e si fermerebbe nella ricerca di altre path valide.

Tutto funziona se digito manualmente l'URL nel browser. Quello che ci serve però è un link da cliccare.

La domanda che sorge spontanea è: come facciamo a creare dei link con parametri query, da inserire all'interno del template costituito dalla lista di treni?

Non posso usare la notazione che si usa nel mondo delle pagine web , ossia aggiungere un parametro query all'URL, come evidenziato qui sotto:

```
<a href="dettaglio/?AFD">Link dettaglio</a>
```

Dobbiamo ricordarci invece della direttiva `RouteLink`, che ora sarà usata nella forma estesa, ossia con l'aggiunta di un array al posto della classica stringa indicativa del percorso:

```
<a [routerLink] = "[PERCORSO, PARAMETRO]"> ... </a>
```

L'array è inserito all'interno delle doppie virgolette, ed è costituito dalle coppie PERCORSO e PARAMETRO, che dovranno essere personalizzate.

Ad esempio per l'applicazione MetroChat, che richiede un link cliccabile all'interno del componente `treni.component.ts`, potrei modificare il template e scrivere:

```
<a [routerLink] = "['inarrivo/dettaglio', metro.idt]"> ... </a>
```

dove `inarrivo/dettaglio` è l'identificativo della path, mentre `metro.idt` è proprio l'identificativo associato ad ogni treno e che potremo sfruttare, in una fase successiva, per recuperare i dati di dettaglio.

Per indirizzare il navigatore allo stesso link, questa volta però usando un 'istruzione inserita nel corpo del componente e non nel template, si sfrutta il metodo `navigate()`, che avevamo visto quando abbiamo parlato di route di Redirect:

```
this.router.navigate([PERCORSO, PARAMETRO]);
```

Il componente `treni.component.ts`, potrebbe quindi diventare:

```
treni/treni.component.ts

import { Metro } from './model/metro.model';
import { Component, OnInit } from '@angular/core';
import { Router } from '@angular/router';

@Component({
  selector: 'ca-treni',
  (continua)
```

```

template: `

<ca-menu></ca-menu>
<h1>Treni in arrivo</h1>
<ca-metro *ngFor="let metro of listametro"
  [inPartenza]="partiti($event)"
  [datiIn]="metro"
  [ora]="'now'"
  (click)="setMetro(metro.idt)">
</ca-metro>
<p>{{trenipartiti}}</p> `
`)

export class TreniComponent implements OnInit {
  listametro: Metro[];
  trenipartiti:string;

  constructor(private router: Router) {
    this.trenipartiti = '';
    this.listametro = [];
    this.now = new Date().getTime();
  }

  ngOnInit() {
    this.listametro = LISTAMETRO;
  }

  setMetro(id: string) {
    this.router.navigate(['inarrivo/dettaglio', id]);
  }

  partiti(id: string) {
    this.trenipartiti += ' | ' + id;
  }
}

```

dove il grosso del cambiamento è avvenuto all'interno della funzione `setMetro()`, in cui il parametro ricevuto in ingresso, rappresentativo dell'identificativo del treno, è stato usato per indirizzare il navigatore ad un URL del tipo:

inarrivo/dettaglio/N

A questo punto Angular, sulla base delle route configurate all'inizio, è in grado di trovare la corrispondenza con l'URL dinamico e visualizzare il componente di dettaglio `dettaglio.component.ts`.

E' chiaro che ora sfruttiamo l'informazione passata tramite l' URL per recuperare tutti i dati del treno, sulla base delle funzionalità messe a disposizione dell'API che si interfaccia con il database dove sono memorizzati i dati.

Queste operazioni le vedremo nel capitolo 12, quando parleremo di recupero dati da una sorgente esterna tramite il protocollo HTTP.

10.12 Recuperare i parametri da un URL

Al fine di visualizzare la pagina dettaglio, dobbiamo prima di tutto imparare a recuperare l'informazione passata tramite l'URL, nel nostro caso l'identificativo del treno.

Eraamo giunti infatti a questo punto:

1. Click sul componente `<ca-metro>` che richiama il metodo `setMetro()`, passando un parametro `id`
2. Apertura di un URL del tipo: `inarrivo/dettaglio/XXX` dove al posto di `XXX` sarà visualizzato l'identificativo associato al treno
3. Intercettazione dell'URL tramite le route con parametri
4. Visualizzazione componente dettaglio

```
setMetro(id: string) {
  // indirizzo l'utente alla sezione dettaglio
  this.router.navigate(['inarrivo/dettaglio', id]);
}
```

Per ora ci limiteremo a visualizzare un messaggio di `alert()`, che mostri l'effettivo parametro recuperato dall'URL, con il quale successivamente interrogheremo l'API tramite una chiamata via HTTP.

Al fine di poter recuperare dall' URL il parametro passato e valorizzare così una proprietà interna al componente, è necessario iniettare nel costruttore della classe del componente `dettagliotreno.component.ts`, un “Service” (vedi capitolo dedicato) - per la precisione `ActivatedRoute` e `ParamMap` – che hanno all'interno una serie di proprietà e metodi (`snapshot` e `get`) che permettono di accedere a tutte le informazioni sulla route.

Una delle sintassi che potremmo usare è la seguente:

```
this.idtreno = oggettoRouter.snapshot.paramMap.get('id')!;
```

Come puoi osservare, abbiamo passato al metodo `get()`, lo stesso identificativo `id` presente nella definizione della route dinamica, mentre `oggettoRouter` è la

variabile privata passata al costruttore, con il meccanismo delle “Dependency Injection”, che vedremo in seguito quando parleremo di “Service”. Il metodo `get()` restituisce un dato di tipo `string` o `null` e questo è il motivo per cui abbiamo inserito il simbolo `!` alla fine (operatore assertivo non-null di TypeScript).

Questa, come dicev amo, è una possibile sintassi, che viene usata quando l'utente necessita di visualizzare solo un'informazione, per poi passare ad altre sezioni dell'applicazione.

Quando parleremo di “Observable”, vedremo che questa modalità di recupero potrà essere modificata e resa più efficiente. Potresti infatti riutilizzare lo stesso componente di dettaglio, per visualizzare più dati in successione, in funzione di un cambio della query ottenuto con dei link interni allo stesso componente genitore.

In questi casi, Angular, ad ogni cambio query, non crea una nuova istanza del componente, ma riutilizza quella creata alla prima chiamata. Questo chiaramente se l'utente rimane nella stessa pagina e non clicca su altre sezioni dell'applicazione.

Giusto per farti vedere anche questa sintassi, dovremo sfruttare l'iscrizione ad un Observable – restituito dalla proprietà `paramMap` – per tenere monitorato ogni eventuale cambiamento dell'URL fatto da azioni interne al componente.

```
oggettoRouter.paramMap.subscribe(params => {
  this.idtreno = params.get('id');
});
```

Tornando al nostro obiettivo, le ulteriori librerie da aggiungere saranno prelevate dallo stesso package di `Router` quindi da `@angular/router`:

```
import { ActivateRoute, ParamMap } from '@angular/router';
```

Il componente `dettaglio.component.ts`, potrà essere creato mettendo insieme tutti questi concetti:

```
treni/dettaglio/dettaglio.component.ts

import { Component, OnInit } from '@angular/core';
import { ActivatedRoute, ParamMap } from '@angular/router';

@Component({
  selector: 'ca-dettaglio',
  template: `<p>ID: {{idtreno}}</p>
            <p>Lista Chat...</p>`
```

(continua)

```

})
}

export class DettaglioComponent implements OnInit {
  idtreno!:string;
  constructor(private route: ActivatedRoute) {
    this.idtreno = this.route.snapshot.paramMap.get('id')!;
    // oppure in alternativa
    // this.route.paramMap.subscribe(params => {
    //   this.idtreno = params.get('id')!;
    // });
    // NB: La Route intercetta eventuali id nulli quindi idtreno
    // sarà sempre valorizzato non null
  }
  ngOnInit() {}
}

```

Navigando all'URL del tipo:

inarrivo/dettaglio/XXX

corrispondente alla path:

inarrivo/dettaglio/:id

sarà recuperato l'id associato al parametro passato tramite l' URL (XXX), che valorizzerà una proprietà `idtreno`, collegata al template del componente. Ricorda che il dato così recuperato è di tipo *string*, quindi eventualmente puoi usare il simbolo + per trasformarlo in numero intero, nel caso ti servisse di questo tipo.

```
this.idtreno = +oggettoRouter.snapshot.paramMap.get('id')!;
```

Con queste righe implementate, ora abbiamo le basi per recuperare un qualsiasi dato da un'API remota.

10.13 Modificare il menu di dettaglio per l'applicazione MetroChat

Ora che abbiamo gran parte delle conoscenze per recuperare informazioni dalle route e per gestire proprietà d'ingresso ai componenti, vediamo come modificare il menu dell'applicazione in modo da diversificarlo non appena l'utente accede alla pagina di dettaglio.

Il risultato finale a cui vogliamo arrivare è simile a questo:

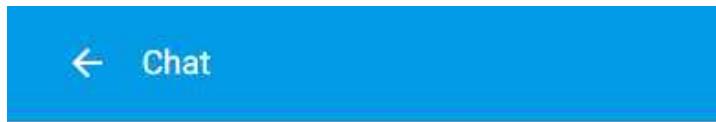


Figura 10.3 – Menu sezione di dettaglio dell'applicazione MetroApp

Il punto di partenza è il componente `menu.component.ts` che avevamo sviluppato nel capitolo 3. Dovremo evidentemente aggiungere una condizione tramite la direttiva `*ngIf` che ci permetta di visualizzare o l'elenco completo dei link, o solo il link per tornare indietro.

Le tecniche insomma che abbiamo già visto con l'esempio delle luci accese e spente.

```
menu/menu.component.html
```

```
<!-- menu applicazione -->
<nav>
  <ul *ngIf="menuback != 'ON'; else showback">
    <li><a routerLink="/inarrivo">{{ link_menu_1 }}</a></li>
    <li><a routerLink="/preferiti">{{ link_menu_2 }}</a></li>
    <li><a routerLink="/login">{{ link_menu_3 }}</a></li>
  </ul>
  <ng-template #showback>
    <ul>
      <li><a routerLink="/inarrivo"> <- Chat </a>
    </ul>
  </ng-template>
</nav>
```

La variabile `menuback` ci permetterà di discriminare tra una versione del menu e l'altra e sarà gestita direttamente nel corpo della classe, grazie alla creazione di una variabile di input che ogni componente dovrà passare:

```
menu/menu.component.ts
```

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

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

export class MenuComponent {
  @Input() menuback!: string;
  link_menu_1:string;
  link_menu_2:string;
```

(continua)

```

link_menu_3:string;

constructor() {
    this.link_menu_1 = 'Treni';
    this.link_menu_2 = 'Preferiti';
    this.link_menu_3 = 'Login';
}
}
}

```

In questo modo, se nel template del componente *dettagliotreno.component.ts* definiamo una variabile valorizzata con la stringa “ON”, “accenderemo” la versione di dettaglio del menu.

treni/dettagliotreno/dettagliotreno.component.ts

```

import { Metro } from '../../../../../model/metro.model';
import { Component, Input } from '@angular/core';

@Component({
  selector: 'ca-dettaglio',
  template:
`<ca-menu [menuback]="menuback"></ca-menu>
<router-outlet></router-outlet>
`)

export class DettagliotrenoComponent {
  menuback:string = '';
  constructor() {
  }
  ngOnInit() {
    this.menuback = 'ON';
  }
}

```

Questa è una possibile soluzione, ma successivamente vedremo altre tecniche per sfruttare direttamente ulteriori dati che è possibile definire nella route.

Capitolo 11

Separare le funzionalità con i Service

11.1 Perché usare i Service

Abbiamo più volte detto che un'applicazione Angular non è altro che un insieme di componenti aventi una struttura ad albero, partendo dal componente radice e via via proseguendo verso quelli interni.

Un principio base per lo sviluppo di un componente è che deve svolgere una **specific mansione**, quindi deve contenere un numero limitato di righe: in poche parole deve essere semplice e sfruttare eventuali funzionalità specifiche richieste per il suo funzionamento, prelevandole da servizi esterni.

Questo nel rispetto del principio visto nel Capitolo 9 detto “**SRP**” (**Single Responsibility Principle**)

Ad esempio, quando un'impresa costruisce un'abitazione, potrebbe decidere di produrre tutto in loco, dai mattoni, alla malta, ai tubi, alle mattonelle etc.

E' chiaro che sarebbe poco efficiente, perché la complessità nel gestire tutte le maestranze, creerebbe troppi colli di bottiglia.

Meglio delegare a servizi esterni e far fare i diversi elementi a ditte specializzate, mentre in loco, limitarsi solo ad assemblarli e fare poche lavorazioni.

Nella figura qui sotto, puoi vedere questi concetti applicati ad un'applicazione Angular costituita da due componenti e da due fornitori di servizi.

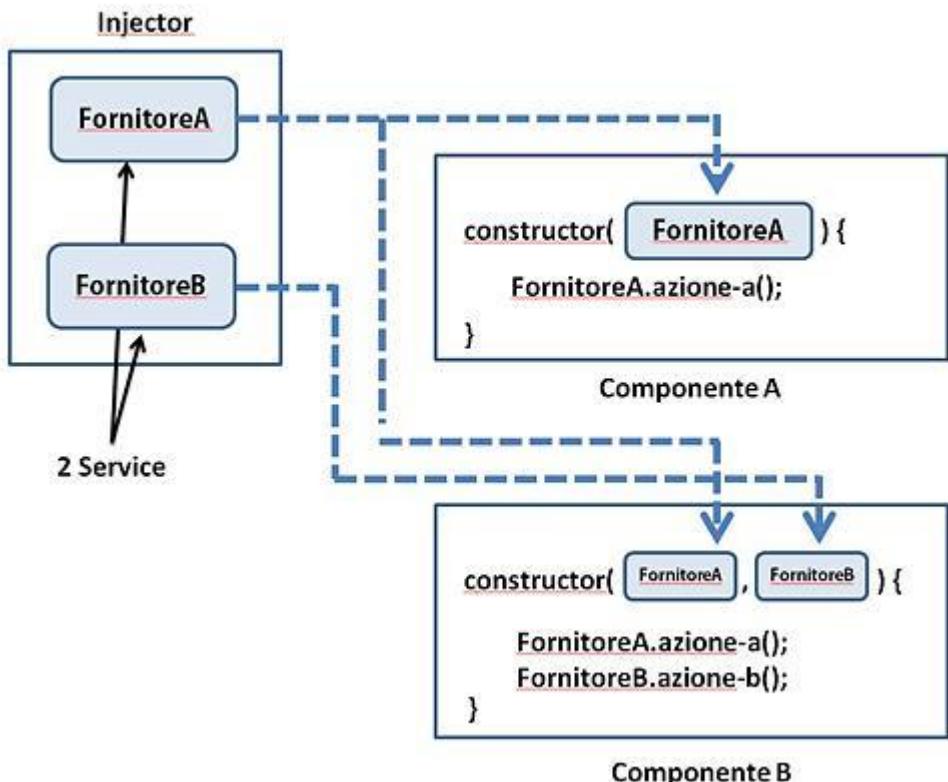


Figura 11.1 – Meccanismo del “Dependency Injection”

In Angular questo concetto si concretizza con la realizzazione di “Service”, o servizi in italiano e tramite il meccanismo chiamato "Dependency Injection" o iniezione delle dipendenze. I Service sono delle classi TypeScript.

Non entreremo nel dettaglio della filosofia di progettazione legata a questi concetti e nemmeno parleremo di quello che succede a livello tecnico, ma ci limiteremo a dire che grazie a questa tecnica, non sarà più compito di ogni componente recuperare i diversi “pezzi” necessari al suo funzionamento, ma tutto sarà centralizzato.

Sarà quindi compito dell'applicazione creare e fornire i “pezzi” ai diversi componenti che li richiederanno.

Centralizzando il processo inoltre, anche tutta la fase di test di un'applicazione viene semplificata, perché basterà cambiare una configurazione a livello centrale per rifletterla a tutta l'applicazione.

Abbiamo già incontrato alcuni esempi d'uso dei Service, quando abbiamo parlato di navigazione e recupero dei parametri da un url. Ora cercheremo di chiarire meglio questi concetti e di applicarli a un caso reale : il **recupero dei dati da un a sorgente esterna**.

In tutti gli esempi visti fino ad ora, che coinvolgevano la visualizzazione dei dati, abbiamo sempre ipotizzato che questi fossero memorizzati all'interno di una costante.

Questa non è certamente una situazione ideale, in quanto i dati non saranno mai delle costanti, ma saranno soggetti a cambiamenti. Inoltre spesso avrà la necessità di effettuare delle operazioni di filtro e manipolazione, per estrarre solo alcune informazioni.

Grazie ai Service e al meccanismo della "Dependency Injection", invece di far fare tutto ad uno specifico componente e ricopiare le stesse funzionalità su altri componenti, si lascia che tutte le operazioni che coinvolgono i dati, siano fatte da una classe specifica, o diverse classi, che poi forniranno il proprio materiale (i dati o le funzionalità) a coloro che li richiederanno.

Il modello che si sfrutta è quello della creazione di una singola istanza, o "SingleTon" come viene detto in gergo tecnico.

Un utilizzo frequente dei Services, è proprio legato a tutte le operazioni di IO (Input e Output).

Cerchiamo allora una soluzione per rendere indipendenti le funzionalità legate a dati, e poterle poi fornire (iniettare) all'occorrenza a chi ce le richiederà.

11.2 Come creare un Service

Tutti i Service che avremo la necessità di creare, sono delle normali classi TypeScript, che per convenzione vengono memorizzate con un nome di file che termina con l'estensione `.service`, sempre con lettere minuscole per evitare problemi nel caso in cui l'applicazione venga erogata da server sensibili ai nomi in maiuscolo e minuscolo.

L'unica differenza rispetto alla classica classe TypeScript è che dovrà contenere un particolare decoratore `@Injectable()` che permetterà ad Angular di "riconoscerla" come classe da poter usare grazie al meccanismo del "Dependency Injection".

Come tutte le classi, che non siano dei componenti, non presenta un template ed è presente la parola `export`, perché la dobbiamo esportare come un modulo.

La sintassi da cui partire per creare un Service sarà:

```
nomeclasse.service.ts

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

@Injectable()
export class NomeClasse {
  constructor(parametri da iniettare) {
  }
}
```

Nota la presenza della riga per importare, dalla libreria `@angular/core`, `Injectable` e nota come il decoratore sia stato inserito facendolo seguire dalle parentesi, senza indicare per ora, alcun argomento.

Lo stesso decoratore `@Component()` è un particolare sottotipo di `@Injectable()`

Un Service può chiaramente avere un costruttore, delle proprietà e dei metodi, esattamente come tutte le classi viste fino ad ora. L'unica differenza sarà la modalità con cui potrà accedere a questi membri dall'esterno, ossia da altri componenti.

Si dovranno applicare gli stessi concetti visti per il service nativo di Angular `ActivatedRoute`, i cui metodi venivano utilizzati solo dopo avere "iniettato" l'oggetto nel costruttore del componente.

Tornando all'esempio dell'app ch e stiamo sviluppando, al fine d'isolare i dati dal codice del componente principale, potremmo creare un Service dedicato proprio al recupero di questi da un database remoto.

Per ora limitiamoci a inserirli manualmente, così come fatto in precedenza, quando abbiamo creato i dati rappresentativi dell'elenco dei treni e dei messaggi scambiati.

Creiamo una costante da esportare di nome `LISTAMETRO`, che salveremo all'interno del file `listametro.ts` nella cartella `dati`:

```
dati/listametro.ts

import { Metro } from './metro/metro.model';
export const LISTAMETRO: Metro[] = [
  {idt:'ASD', linea:'Rossa', numchatting:23, tempo:500},
  {idt:'AKE', linea:'Verde', numchatting:33, tempo:900},
  {idt:'BFD', linea:'Gialla', numchatting:13, tempo:1500}
];
```

Vediamo anche un possibile esempio di sviluppo del Service, che potremmo salvare con nome `treni.service.ts` nella cartella `service` interna alla cartella principale `app`:

```
service/treni.service.ts
```

```
import { Injectable } from '@angular/core';
import { Metro } from './.../metro.model';
import { LISTAMETRO } from './.../dati/listametro';

@Injectable() // NB:non ho ancora indicato chi creerà il service
export class TreniService {
  constructor() {}

  getListaMetro(): Metro[] {
    // recupero i dati statici per ora
    return LISTAMETRO;
  }

  getDettaglioMetro(id: string): Metro {
    // recupero uno specifico elemento dell'array
  }
}
```

Come puoi osservare, per ora il costruttore non dipende da altri Service, quindi non ha delle dipendenze, e sono presenti solo due metodi, `getListaMetro()` e `getDettaglioMetro()`, usati rispettivamente per recuperare la lista completa dei treni e il dettaglio, sempre sfruttando dei dati statici già inseriti nell'app e che dovranno essere sostituiti con dati recuperati in tempo reale.

Con il progredire delle funzionalità dell'app, potrà aggiungere altri metodi specifici, come, ad esempio, la memorizzazione dei treni preferiti.

Fai attenzione che, come per tutte le classi, fino a quando non creo l'istanza di questa, non succederà nulla. La domanda da farsi è quindi: chi crea l'istanza di un service e come? Vediamo dunque l'ultimo passaggio da fare per poter sfruttare i metodi interni alla classe `TreniService`.

11.3 Registrare un Service

Definire una classe non serve a molto se poi non si crea l'istanza di questa. Fortunatamente in Angular non ci dobbiamo preoccupare di questa operazione, perché viene fatta in automatico, a patto di effettuare la cosiddetta “**registrazione di un Service**”.

Abbiamo infatti detto che lo scopo dei Service è quello di semplificare le mansioni di un componente, andando a creare delle funzionalità gestite in modo centralizzato,

che all'occorrenza vengono fornite, tramite un processo di “iniezione”, ai diversi componenti di un'applicazione.

A partire dalla versione 6.0 di Angular, sarà sufficiente aggiungere al decoratore `@Injectable`, un oggetto (metadato) con la chiave `providerIn`, valorizzata con una stringa che identifica chi dovrà creare il Service.

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

Con la stringa ‘root’, demandiamo all’*Application Injector* l’onere di creare il Service. In questo modo sarà disponibile per tutti i componenti dell’applicazione tramite il meccanismo dell’ “Iniezione delle dipendenze” (“Dependency Injection”), a patto di importarlo.

Nelle versioni di Angular antecedenti alla 6.0, si poteva omettere questo metadato, ma era necessario fare delle modifiche al file `app.module.ts`. In rete troverai molti esempi di codice che sfruttano ancora questa sintassi, quindi è bene conoscerla.

Si aggiunge il nome del Service alla proprietà `providers` del decoratore `@NgModule()` presente nel modulo radice dell’applicazione. Ad esempio, nell’ipotesi tu abbia due Service di nome `fornitoreA` e `fornitoreB`, scriveremo:

```
app.module.ts

import { fornitoreA, fornitoreB } from '/servizi';

@NgModule({
  declarations: [MioComponent],
  provider: [fornitoreA, fornitoreB]
})
```

All’interno della proprietà `provider`, si definisce un array con l’elenco dei “fornitori” da usare. Questa sintassi è ancora ammessa nella versione 6 e successive, anche se noi useremo quella che sfrutta la proprietà `providerIn`. Per velocizzare la creazione di un service, si può sfruttare direttamente la linea di comando CLI, che creerà gran parte del codice visto sopra, direttamente all’interno della cartella `service`:

```
ng g service service/treni
```

11.4 Come usare un Service

All'interno del componente che ha la necessità di usare le funzionalità del Service, si aggiunge al costruttore un argomento, corrispondente ad una proprietà privata (`private`) di tipo corrispondente al Service (nome della classe) che necessito.

Ad esempio, se avessi creato un Service con nome `fornitoreA`, e questo mi servisse nel componente `Cucina`, potrei scrivere:

```
cucina.component.ts

import { fornitoreA } from './fornitoreA.service';
@Component({
  selector: 'cucina',
  templateUrl: '...'
})
export CucinaComponent {
  constructor(private pavimenti:fornitoreA) {
    // ora posso usare il service con this.pavimenti
  }
}
```

Al costruttore della classe, ho aggiunto un argomento di nome `pavimenti` indicando come tipo di dato, il nome scelto per il Service (`fornitore A`).

Affinché Angular sia in grado di recuperare la definizione di `fornitoreA`, lo dovrò importare con la classica istruzione di `import`.

Una volta che il service è reso disponibile al componente, posso utilizzarne i diversi metodi definiti all'interno che lavorano sull'istanza originaria (singola).

Questa istanza è quella che si crea grazie all'indicazione del *provider*, inserito direttamente come argomento al decoratore `@Injection`. Questo, ripeto, se adottiamo la sintassi introdotta a partire dalla versione 6.0 di Angular.

11.5 Gestire Dati Remoti con i Service

Tornando all'esempio dell'app MetroChat, modifichiamo il codice visto in precedenza in modo da aggiungerci anche l'oggetto con la proprietà `providerIn`.

NB: Questa operazione non è necessaria se hai creato il service sfruttando la linea di comando.

service/treni.service.ts

```

import { Injectable } from '@angular/core';
import { Metro } from '../../model/metro.model';
import { LISTAMETRO } from '../../dati/listametro';

@Injectable({
  providerIn: 'root'
})
export class TreniService {
  constructor() {}
  getListaMetro(): Metro[] {
    // recupero i dati statici per ora
    return LISTAMETRO;
  }
  getDettaglioMetro(id:string): Metro {
    // recupero uno specifico elemento dell'array
  }
}

```

A questo punto, siamo pronti per poter sfruttare il meccanismo del "Dependency Injection", per iniettare il Service all'interno del componente responsabile della visualizzazione della lista dei treni in arrivo.

treni/treni.component.ts

```

import { Component, OnInit } from '@angular/core';
import { Router } from '@angular/router';
import { Metro } from '../../model/metro.model';
import { TreniService } from '../../service/treni.service';

@Component({
  selector: 'ca-treni',
  template: `
<ca-menu></ca-menu>
<h1>Treni in Arrivo</h1>
<ca-metro *ngFor="let metro of listametro"
  (inPartenza)="partiti($event)"
  [datiIn]="metro"
  [ora]="'now'"
  (click)="setMetro(metro.idt)">
</ca-metro><p>{{trenipartiti}}</p>
`})
export class TreniComponent implements OnInit {
  listametro!: Metro[];
  trenipartiti!:string;
  now!:number;
  constructor(private router: Router, private treniservice: TreniService) {
    this.trenipartiti = '';
    this.listametro = [];
    this.now = new Date().getTime();
  }
}

```

(continua)

```

ngOnInit() {
    this.listametro = this.treniservice.getListametro();
}

setMetro(id:string) {
    this.router.navigate(['inarrivo/dettaglio', id]);
}

partiti(id:string) {
    this.trenipartiti += ' | ' + id;
}
}

```

All'interno di `ngOnInit()`, ora potrò richiamare il metodo del Service, che abbiamo provveduto ad iniettare nel costruttore della classe, tramite una variabile privata di nome `treniservice`.

Il costruttore ora ha due argomenti, uno usato per accedere a `Router` e l'altro per accedere a `TreniService`.

Visto che il metodo del Service `getListMetro()` restituisce proprio un array di oggetti Metro, posso usarlo direttamente per valorizzare la proprietà `listametro`.

Analogo discorso per il componente di dettaglio, che anch'esso avrà bisogno del Service, per sfruttare il metodo `getDettaglioMetro(id)` al fine di recuperare i parametri del treno sulla base dell'identificativo passato.

```

treni/dettagliotreno/dettaglio.component.ts

// ....
import { Metro } from './../../../../model/metro.model';
import { ActivatedRoute, ParamMap } from '@angular/router';
import { TreniService } from './../../../../service/treni.service';

@Component({
selector: 'ca-dettaglio',
template: `<div *ngIf="treno">
    <p> Treno Linea: {{treno.linea}} </p>
    <p> ID: {{treno.idt}}</p>
</div> `
})

export class DettaglioComponent implements OnInit {
    idtreno!: string;
    treno!: Metro;
    constructor(private route: ActivatedRoute,
                private treniservice: TreniService) {
    }
}

```

(continua)

```

ngOnInit() {
    this.idtreno = this.route.snapshot.paramMap.get('id')!;
    this.getDettaglioMetro(this.idtreno);
}

// Richiamo il metodo getDettaglioMetro() del Service
getDettaglioMetro(idtr:string) {
    this.treno = this.treniservice.getDettaglioMetro(idtr);
}
}

```

Dovendo recuperare dei dati, in modalità asincrona, da una sorgente remota caratterizzata da un certo tempo di latenza, è preferibile spostare le righe interne al costruttore, all'interno del metodo `ngOnInit()`.

Per comodità ho creato un metodo interno con lo stesso nome di quello presente nel service - `getDettaglioMetro()` -, a cui ho passato il parametro recuperato dall'url e rappresentativo dell'identificativo del treno selezionato.

Il metodo `getDettaglioMetro()` del Service invece, restituisce un oggetto di tipo `Metro`, che userò per valorizzare la proprietà `treno` di tipo `Metro`.

Inoltre dovrò accertarmi che, nel momento in cui verrà visualizzato il template, la variabile `treno` non sia nulla, questo per evitare che Angular segnali un errore legato alla mancanza dell'oggetto.

Posso agevolmente sfruttare la direttiva `*ngIf`:

```

template: `<div *ngIf="treno">
    <p> Treno Linea: {{treno.linea}} </p>
    <p> ID: {{treno.idt}}</p>
</div>
<h2>Lista Chat</h2>`

```

Il passo successivo, per liberarci definitivamente dei dati statici memorizzati all'interno del file `datiremoti.ts`, è quello di scoprire cosa mette a disposizione Angular, per gestire l'accesso a sorgenti d'informazioni memorizzate su servizi remoti.

Capitolo 12

Accedere a dati remoti

12.1 Manipolare i dati via HTTP

Un'applicazione che non sia in grado di interagire con l'utente e di rispondere alle richieste, è come una macchina senza motore: serve a poco.

Noi non ci preoccuperemo di realizzare un'applicazione stile WhatsApp, quindi un'app in grado di comunicare in tempo reale ; cercheremo di gettare le basi per capire come avviene lo scambio d 'informazioni tra un'applicazione Angular e un ipotetico servizio remoto, ad esempio un'API in grado d'interrogare un database e di scriverci all'interno.

Attraverso queste basi, potrai poi proseguire per creare la tua applicazione in tempo reale, sfruttando ad esempio servizi come "Firebase".

Il primo passo per poter comunicare con API raggiungibili tramite la rete internet sfruttando il protocollo HTTP, è **configurare** l'applicazione per sfruttare una serie di servizi non inclusi nei moduli standard di Angular.

Questi servizi sono già stati creati e per poterli sfruttare con il meccanismo del "Dependency Injection", dovremo importare un modulo, invece di usare la proprietà `provider`, come visto per i classici Service.

Il primo passo sarà quello modificare la proprietà `import` del decoratore `@NgModule()` presente nell'`AppModule`, in modo che in automatico venga effettuata la registrazione del service che ci servirà per comunicare via HTTP.

A partire dalla versione 4.3 di Angular, è stata introdotta una nuova classe per gestire tutte le richieste da inviare in rete tramite HTTP. Questa classe prende il nome di `HttpClient`, e sostituisce la precedente `Http`. Sono state aggiunte diverse funzionalità, ma noi vedremo solo le principali, finalizzate al nostro obiettivo.

Le due classi in realtà sono molto simili e potrebbero essere usate in modo indistinto per le necessità del nostro progetto, ma chiaramente basiamoci su questa versione potenziata.

Analizziamo allora le modifiche da apportare al codice del file `app.module.ts` del progetto, al fine di poter interrogare API remote e inviare dati via HTTP.

```

app.module.ts

// ...
import { BrowserModule } from '@angular/platform-browser';

// 1) Importazione di HttpClientModule
import { HttpClientModule } from '@angular/common/http';
...

@NgModule({
  declarations: [
    //...
  ],
  // 2) Aggiunta di HttpClientModule alla proprietà
  imports: [
    BrowserModule,
    HttpClientModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Tralasciando le sezioni già viste e indicandole con ... dei puntini di sospensione ..., possiamo notare:

- 1) la classica riga per importare un modulo, che è `HttpClientModule` dal package `@angular/common/http`, e poi
- 2) l'inserimento di questo, all'interno della proprietà `imports`.

E' importante osservare anche l'ordine in cui è stata inserita la riga all'interno di `imports`. Questa infatti deve seguire quella relativa all'importazione del modulo `BrowserModule`.

Aggiunte queste due righe al file, possiamo procedere con l'utilizzo del Service **HttpClient**, essendo ora disponibile per tutti i componenti dell'app, tramite il meccanismo della "Dependency Injection".

Questo Service ci permetterà di accedere alla rete sfruttando **XMLHttpRequest (XHR)**.

In questo modo elimineremo definitivamente il file locale sfruttato in tutti gli esempi visti fino ad ora.

Le modifiche che dovremo fare , pertanto, riguarderanno il Service *treni.service.ts* e in futuro anche il Service che svilupperemo per recuperare la lista dei messaggi della chat (*chat.service.ts*).

12.2 Recuperare dati in GET

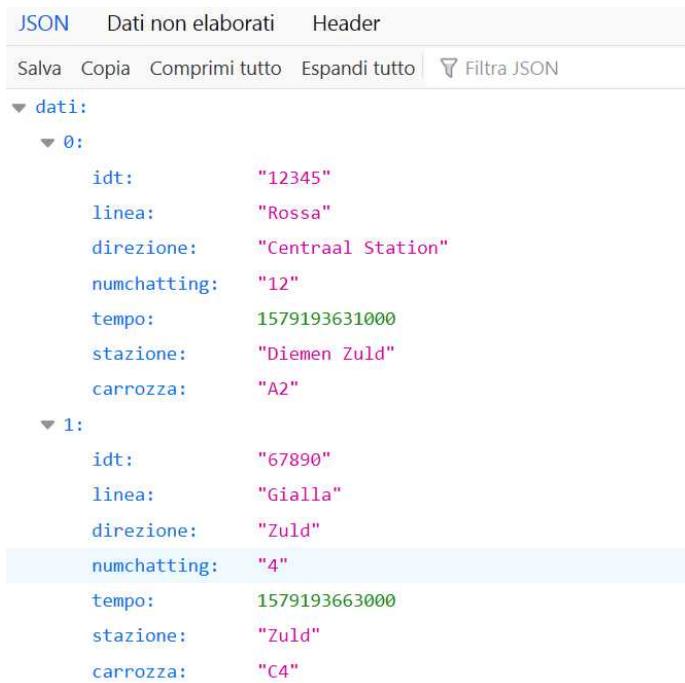
La prima domanda da farsi è : che forma avranno i dati restituiti dal servizio remoto che andremo a interrogare?

Nella quasi totalità dei casi, dovremo manipolare dei dati rappresentati nel formato JSON, quindi la prima ipotesi che faremo è che i dati dei diversi treni in arrivo, siano rappresentati con la notazione JSON costituita da una chiave/proprietà di nome *dati*, con all'interno un array di oggetti Metro:

```
{
  "dati": [
    {
      "idt": "12345",
      "linea": "Rossa",
      "direzione": "Centraal Station",
      "numchatting": "12",
      "tempo": "1488053502000",
      "stazione": "Diemen Zuld"
    },
    {
      "idt": "34523",
      "linea": "Gialla",
      "direzione": "Centraal Station",
      "numchatting": "23",
      "tempo": "1498053534000",
      "stazione": "Zuld"
    }
  ]
}
```

In questo modo potrò sfruttare proprio la chiave *dati* per scorrere tra tutti gli elementi dell'array e visualizzare così l'elenco dei treni.

Se tu provassi ad accedere all'url dell'API adibita al recupero di questo elenco (vedi appendice), visualizzeresti proprio il dettaglio di due treni:



The screenshot shows a JSON viewer with the following structure:

```

{
  "dati": [
    {
      "idt": "12345",
      "linea": "Rossa",
      "direzione": "Centraal Station",
      "numchatting": "12",
      "tempo": 1579193631000,
      "stazione": "Diemen Zuld",
      "carrozza": "A2"
    },
    {
      "idt": "67890",
      "linea": "Gialla",
      "direzione": "Zuld",
      "numchatting": "4",
      "tempo": 1579193663000,
      "stazione": "Zuld",
      "carrozza": "C4"
    }
  ]
}

```

Figura 12.1 – Rappresentazione JSON dei singoli treni restituiti dall’API

Nell’ipotesi il server remoto sia dotato di interprete PHP, potremmo abbozzare un programma in grado di restituire tale stringa, scrivendo:

```

metro/index.php

//...
// preparo la connessione ad database
// e interrogo la tabella ipotetica treni
while($row = mysqli_fetch_array($query)) {
  $results['idt'] = $row['idtreno'];
  $results['linea'] = $row['linea'];
  $results['direzione'] = $row['direzione'];
  $results['numchatting'] = $row['numchatting'];
  $results['tempo'] = $row['unixtime'];
  $results['stazione'] = $row['stazione'];
  array_push($strout, $results);
}
header('Content-Type: application/json');
echo '{"dati":"' . json_encode($strout) . '}';

```

dove ho supposto che tutti i dati siano memorizzati all’interno di un database MySQL. Analogi discorsi potremmo fare con Node.

Per poter accedere a dati remoti via HTTP, Angular fornisce il metodo `get()`, presente all'interno del Service `HttpClient`, che dovrà essere iniettato nel costruttore della classe. Il metodo ha diverse “firme”, ma noi useremo quella standard, che prevede il passaggio del solo parametro obbligatorio `URL`, visto che i dati da recuperare sono già nella forma predefinita ossia JSON.

A livello predefinito il metodo restituisce solo il `body` della risposta e non tutte le intestazioni HTTP. Chiaramente è possibile recuperare tutte, aggiungendo ulteriori parametri nella richiesta come `{observe : "response"}`.

Per la lista completa di tutte le firme ammesse per il metodo, consulta questo link: <https://angular.io/api/common/http/HttpClient#get>

La tipica sintassi di una richiesta HTTP in modalità GET, prevede il concatenamento di due metodi: `get()` e `pipe()`. Quest'ultimo è un operatore che fa parte della libreria RxJS (ReactiveX for JavaScript), che permette di eseguire in successione più funzioni, senza usare la classica notazione della concatenazione:

```
oggettoHttp.get<TIPO>(URL, OPZIONI) // restituisce un Observable
    .pipe(
        map(),
        catchError()
    );

```

URL è l'indirizzo dell'API da interrogare, mentre `map()` è anch'esso un operatore della libreria RxJS, che permette di eseguire una trasformazione su ogni singolo dato dell' Observable, proprio quello restituito da `get()`. Ad esempio potremmo estrarre solo una certa chiave dalla risposta `body` fornita in formato JSON.

Essendo un dato Observable, al fine di attivare la richiesta effettiva e riuscire così a leggere i dati, dovremo effettuare l'operazione di `subscribe()`. Capiremo meglio questi concetti in uno dei prossimi capitoli.

A livello predefinito, il `body` della risposta fornito da `http.get()`, è un oggetto JSON a cui Angular non associa un tipo di dato specifico. Questa associazione dovrà essere fatta da noi, usando la classica notazione `<TIPO>` (es. `<Metro[]>`), sulla base delle scelte progettuali con cui è stata realizzata l'API.

Come dicevamo, è possibile indicare ulteriori informazioni HTTP opzionali da inviare in ogni richiesta. Ad esempio, se dobbiamo interpretare il corpo della risposta come se fosse del normale testo (es. una pagina web con i relativi tag), possiamo aggiungere un ulteriore parametro:

```
oggettoHttp.get(URL, { responseType: "text" })
//...
```

Al fine d'intercettare eventuali errori di comunicazione, è presente l'operatore RxJS `catchError()`, che ci permetterà di gestirli direttamente nel corpo del `subscribe()`, perché anche `catchError()` restituirà un Observable.

Applicando questi concetti al Service `metro.service.ts` e ipotizzando di creare due nuovi metodi, il primo `getListaMetroObservable()`, da usare al posto del precedente `getListaMetro()` e il secondo `getDettaglioMetroObservable()` da usare al posto di `getDettaglioMetro()`, otterremo il seguente codice:

```
service/metro.service.ts

//...
// 1
import { HttpClient } from '@angular/common/http';
import { throwError, Observable } from 'rxjs';
import { catchError, map, tap } from 'rxjs/operators';

@Injectable()
export class MetroService {

// 2
private apiUrl = 'https://www.dcopelli.it/test/angular/metro';

// 3
constructor(private http: HttpClient) {}

getListaMetroObservable(): Observable<Metro[]> {
    return this.http.get<Metro[]>(this.apiUrl)
        .pipe(
            map((risposta:any) => risposta['dati']),
            catchError(this.handleErrorObs)
        );
}

getDettaglioMetroObservable(idt: string): Observable<Metro> {
    return this.http.get<Metro>(this.apiUrl + '?idt=' + idt)
        .pipe(
            map((risposta:any) => risposta['dati']),
            catchError(this.handleErrorObs)
        );
}

private handleErrorObs(error: any) {
    return throwError(error.message || error);
}
}
```

Per il recupero i dati di dettaglio del treno, l'interrogazione dell'API dovrà essere fatta aggiungendo all'URL un parametro query, rappresentativo dell'id identificativo di ogni treno. Una possibile tecnica è quella di creare “manualmente” la stringa finale:

```
this.apiUrl + '?idt=' + idt // oppure usare i backtick
```

In questo modo, per ogni identificativo `idt`, otterremo un URL dinamico del tipo:

```
https://www.dcopelli.it/test/angular/metro/?idt=XXX
```

Per poter accedere ai dati effettivi, dovremo effettuare il `subscribe()` ai due metodi, quindi modificare il componente `dettaglio.component.ts`, come vedremo successivamente.

Altre sezioni interessanti da commentare sono il punto //1 che ci permette di importare sia il tipo Observable, sia il gestore degli errori dalla libreria RxJS, sia gli operatori dalla libreria RxJS da `rxjs/operators`, e il punto //3 che ci permette d'iniettare nel costruttore il Service `HttpClient`.

Analizzando in dettaglio il metodo `getListaMetroObservable()`, vediamo che restituisce un Observable, a cui abbiamo effettuato il casting assegnandogli il tipo `Metro[]`, da qui la notazione:

```
getListaMetroObservable(): Observable<Metro[]> { ... }
```

Questo perché, come già detto più volte, `http.get()` nella forma che useremo noi, restituisce sempre un Observable rappresentato da un oggetto JSON senza un tipo specifico e costituito solo dalla sezione `body` della risposta fornita dal server. Tale oggetto ci servirà per valorizzare la variabile locale di nome `risposta`.

Nel nostro caso, l'informazione da prelevare restituita dall'API, si trova all'interno della chiave `dati` (*figura 12.2.1*), quindi utilizzeremo la notazione con le parentesi quadre, e per evitare errori di compilazione, indicheremo a TypeScript che il dato è di tipo generico `any`:

```
map((risposta:any) => risposta['dati'])
```

In alternativa, avremmo potuto creare un'interfaccia per definire il tipo della risposta, potendo poi sfruttare la notazione del punto.

L'array di oggetti `i Metro` così recuperato, dovrà avere la stessa sequenza di dati progettati nel modello interno all'applicazione, ma chiaramente queste informazioni sono stabilite da chi ha realizzato l'API.

Analizzando la *figura 12.2.1* vista all'inizio del capitolo, scoprirai che ci sono due chiavi aggiuntive non previste nel modello creato in precedenza.

Pertanto dovremo integrare questo modello, andando ad aggiungere i campi mancanti `direzione` e `stazione`, ma lascio a te questo facile esercizio.

Nel caso ci sia un errore di comunicazione o una risposta negativa del server, tramite `cacthError()` riusciamo a eseguire un metodo interno al Service, che recupera l'oggetto di tipo `HttpErrorResponse` e accede a proprietà come `message`, che contiene proprio il messaggio d'errore inviato dal server remoto.

Dovendo restituire un `Observable`, dovremo usare il metodo `throwError()` di RxJS per comunicare a `subscribe()` l'informazione da mostrare in caso di errore : il messaggio, nel caso sia presente, oppure l'oggetto completo. Chiaramente lo dovrò importare dalla libreria RxJS.

```
return throwError(err.message || err.error);
```

Come dicevamo, il passo successivo sarà quello d'integrare i metodi del Service, sia all'interno del componente `treni.component.ts`, che all'interno del componente `dettaglio.component.ts`

Per il primo, le modifiche riguarderanno la creazione di un metodo per effettuare il `subscribe()` all'`Observable` che restituisce la lista dei treni:

```
treni/treni.component.ts
// ...

@Component({
  selector: 'ca-treni',
  template: `
    <ca-menu></ca-menu>
    <h1>Treni in Arrivo</h1>
    <div class="listtreni">
      <ca-metro *ngFor="let metro of listametro"
        (in partenza)="partiti($event)" (continua)
```

```

[datiIn]="metro"
[ora]="now"
(click)="setMetro(metro.idt)">
</ca-metro>
<div *ngIf="errormsg">{{errormsg}}</div>
</div>
<p>{{trenipartiti}}</p>
`)

export class TreniComponent implements OnInit {
  listametro: Metro[];
  trenipartiti: string;
  errormsg!: any;
  constructor(private router: Router, private treniservice: TreniService) {
    this.trenipartiti = '';
    this.listametro = [];
    this.now = new Date().getTime();
  }

//1
ngOnInit() {
  this.getListametroObservable();
}

//2
getListametroObservable() {
  this.treniservice.getListametroObservable()
    .subscribe(
      risp => this.listametro = risp,
      error => this.errormsg = error
    );
}

setMetro(id: string) {
  this.router.navigate(['inarrivo/dettaglio', id]);
}

partiti(id: string) {
  this.trenipartiti += " | " + id;
}
}

```

Nel punto //1, all'interno di `ngOnInit()` abbiamo richiamato un metodo definito internamente alla classe, che dovrà effettuare il `subscribe()` all'Observable restituito da `getListametroObservable()`.

In questo modo, potremmo usare tale valore per valorizzare l'array `listametro` con oggetti `Metro` da collegare all'interno del template del componente.

```
risp => this.listametro = risp
```

In caso di errore, sarà valorizzata una proprietà `errmsg`, da mostrare nel template, con una classica direttiva `*ngIf`.

```
error => this errmsg = error
```

Se ora proviamo l'applicazione, utilizzando il link del server remoto visto all'inizio del capitolo, questo è l'errore che con molta probabilità ti sarà visualizzato:



✖ XMLHttpRequest cannot load <http://www.dcopelli.it/test/angular/metro>. Redirect from '<http://www.dcopelli.it/test/angular/metro>' to '<http://www.dcopelli.it/test/angular/metro/>' has been blocked by CORS policy: No 'Access-Control-Allow-Origin' header is present on the requested resource. Origin '<http://localhost:4200>' is therefore not allowed access.

Figura 12.2 – Errore legato a una richiesta di una risorsa esterna al dominio dell'app

Questo perché il browser, per ragioni di sicurezza, impedisce di effettuare delle chiamate esterne al proprio computer. Se stai usando Chrome, puoi scaricare una delle tante estensioni, che permettono di bypassare questa sicurezza.

Devi cercare "Allow-Control-Origin" su Chrome Web Store e installarne una per poi abilitarla all'occorrenza. Per maggiori informazioni puoi consultare questo articolo: https://www.video-corsi.com/creareapp/debug_app_in_chrome_e_cross_origin_policy.htm

Per il componente di dettaglio `dettaglio.component.ts`, si sfruttano le stesse considerazioni fatte in precedenza:

```
treni/dettagliotreno/dettaglio.component.ts
//...
import { Metro } from '../../../../../model/metro.model';
import { TreniService } from '../../../../../service/treni.service';

@Component({
  selector: 'ca-dettaglio',
  template: `<div *ngIf="treno">
    <p> Treno Linea: {{treno.linea}} </p>
    (continua)
```

```

        <p> ID: {{treno.idt}}</p>
    </div>
    <h2>Lista Chat</h2>
    `

})

export class DettaglioComponent implements OnInit {
    idtreno!: string;
    treno!: Metro;
    errormsg!: any;

    constructor(private route: ActivatedRoute,
                private treniservice:TreniService) {
    }

    ngOnInit() {
        this.idtreno = this.route.snapshot.paramMap.get('id')!;
        // 1)
        this.getDettaglioMetroObservable(this.idtreno);
    }

    getDettaglioMetroObservable(idt: string) {
        this.treniservice.getDettaglioMetroObservable(idt)
            .subscribe(
                (risp:any) => this.treno = risp[0],
                error => this.errormsg = error
            );
    }

    getDettaglioMetro(idtr: string) {
        this.treno = this.treniservice.getDettaglioMetro(idtr);
    }
}

```

In questo caso, il valore restituito dall'API, è un array con un singolo oggetto Metro, come possiamo visualizzare nella *figura 12.3*, ottenuta sfruttando la query con `idt` pari al valore “12345”:

The screenshot shows a JSON viewer interface with the URL <https://www.dcopelli.it/test/angular/metro/?idt=12345>. The JSON structure is as follows:

```

{
  "dati": [
    {
      "idt": "12345",
      "linea": "Rossa",
      "direzione": "Centraal Station",
      "numchatting": "12",
      "tempo": 1578589733000,
      "stazione": "Diemen Zuid",
      "carrozza": "A2"
    }
  ]
}
  
```

Figura 12.3 – Rappresentazione JSON del singolo treno con identificativo idt=12345

Pertanto dovremo valorizzare la variabile `treno` solo con il primo elemento dell'array:

```
(risp:any) => this.treno = risp[0]
```

12.3 Inviare dati in modalità POST

Non appena avremo la necessità d'inviare delle informazioni da memorizzare, cancellare, aggiornare, su un server remoto, entra in gioco la modalità POST, che a sua volta può essere suddivisa nella modalità PUT e DELETE per le quali Angular ha previsto dei metodi dedicati.

Senza scomodare troppo la fantasia e senza perdere tempo in chiacchiere, la sintassi da usare per interrogare una sorgente di dati esterna inviando delle informazioni dall'app Angular, è leggermente più complessa rispetto a GET, perché entra in gioco anche la tipologia di intestazione da inviare al server.

In Angular è previsto il metodo `post()` del Service HTTP, che come detto per il metodo `get()`, presenta diverse firme. Noi useremo quella base, che ci permetterà di inviare e ricevere i dati considerandoli nel formato JSON :

```
oggettoHttp.post(URL, DATI)
```

Come già visto per `get()`, anche `post()` restituisce un Observable, quindi per dare avvio al recupero del dato effettivo, dovremo effettuare un `subscribe()`.

Anche in questo caso nulla di complesso , tranne alcune precisazioni su come confezionare sia i dati, sia le intestazioni da inviare, nel caso tu debba specificare un formato diverso da quello predefinito JSON.

Analizziamo in dettaglio i diversi argomenti da passare al metodo:

- 1) URL: stringa rappresentativa dell'url a cui inviare i dati.
- 2) DATI: nel nostro caso confezioneremo i dati come se fossero un oggetto literal JavaScript, quindi nell'ipotesi d'inviare un solo campo *id* valorizzato con la stringa 'ABC', dovremmo scrivere:

```
{ id: 'ABC' }
```

Non sempre i dati da inviare e ricevere sono nel formato JSON, quindi è possibile usare anche la firma estesa del metodo , che prevede un terzo parametro costituito da un oggetto JavaScript con una o più proprietà:

```
oggettoHttp.post(URL, DATI, OPZIONI)
```

In quest'ultimo, si possono specificare le intestazioni, il tipo di risposta che si vuole ricevere, dei parametri aggiuntivi da inviare nell'URL e tenere traccia dell'andamento di operazioni come l'upload di un file o il relativo download.

Ad esempio, nel caso tu voglia impostare delle intestazioni HTTP particolari, dovrà creare un oggetto JavaScript inserendo la proprietà `headers`, valorizzata con un oggetto di tipo `Httpheaders`.

Sfrutteremo la classe `HttpHeaders` - che dovrà essere importata da `@angular/common/http` - e tramite il metodo `set()`, imposteremo il tipo di intestazione richiesto dall'API remota ed eventualmente altri dati.

Ad esempio, nel caso volessimo impostare il „Content-Type” della richiesta, dovremo scrivere:

```
new HttpHeaders().set('Content-Type', 'application/TIPO');
```

dove TIPO è il tipo del formato di dato da inviare (es. json,xml,text, etc.)

L’informazione legata alle intestazioni è di fondamentale importanza quando nel server remoto è installato PHP, in quanto i dati inviati in POST sono tipicamente interpretati come se arrivassero da un modulo web, quindi con un „Content-type” impostato a "application/x-www-form-urlencoded" e non sarà possibile usare l’istruzione `$_POST['nomecampo']` per il recupero dei campi , se il tipo di „Content-type” non è corretto.

Il metodo `post()`, nella prima forma vista all’inizio, invierà i dati impostando in automatico il „Content-type” al tipo JSON, quindi in alternativa, potremmo creare l’applicazione PHP di recupero dati, in modo che siano letti da `php://input`, bypassando così il problema dell’uso di `$_POST[]`.

Giusto per vedere un esempio completo, queste potrebbero essere le righe per impostare manualmente la tipologia di contenuto da inviare:

```
// 1) Oltre a HttpClient importo HttpHeaders
import {HttpClient, HttpHeaders } from '@angular/common/http';
...
// 2) Creo le intestazioni
instestaz = new HttpHeaders().set('Content-Type','application/json');
...
// 3) Le aggiungo come terzo parametro OPZIONALE
oggettoHttp.post(URL, {'id': 'ABC'}, {'headers':intestaz})
```

Ora che abbiamo visto gli usi più comuni del metodo `post()`, cerchiamo di applicarli all’applicazione MetroChat. Il primo passo è quello di capire quali dati deve inviare l’applicazione.

Per poter fare questo, dobbiamo aggiungere una nuova sezione, quella che permetterà all’utente d’inserire i propri messaggi, nella chat di ogni treno.

L’immagine di *figura 12.4*, rappresenta proprio il risultato finale che ci siamo prefissati di raggiungere da qui in avanti.

E' la "pagina" che l'utente vedrà dopo aver cliccato su uno dei treni visualizzati nella schermata principale dell'applicazione.

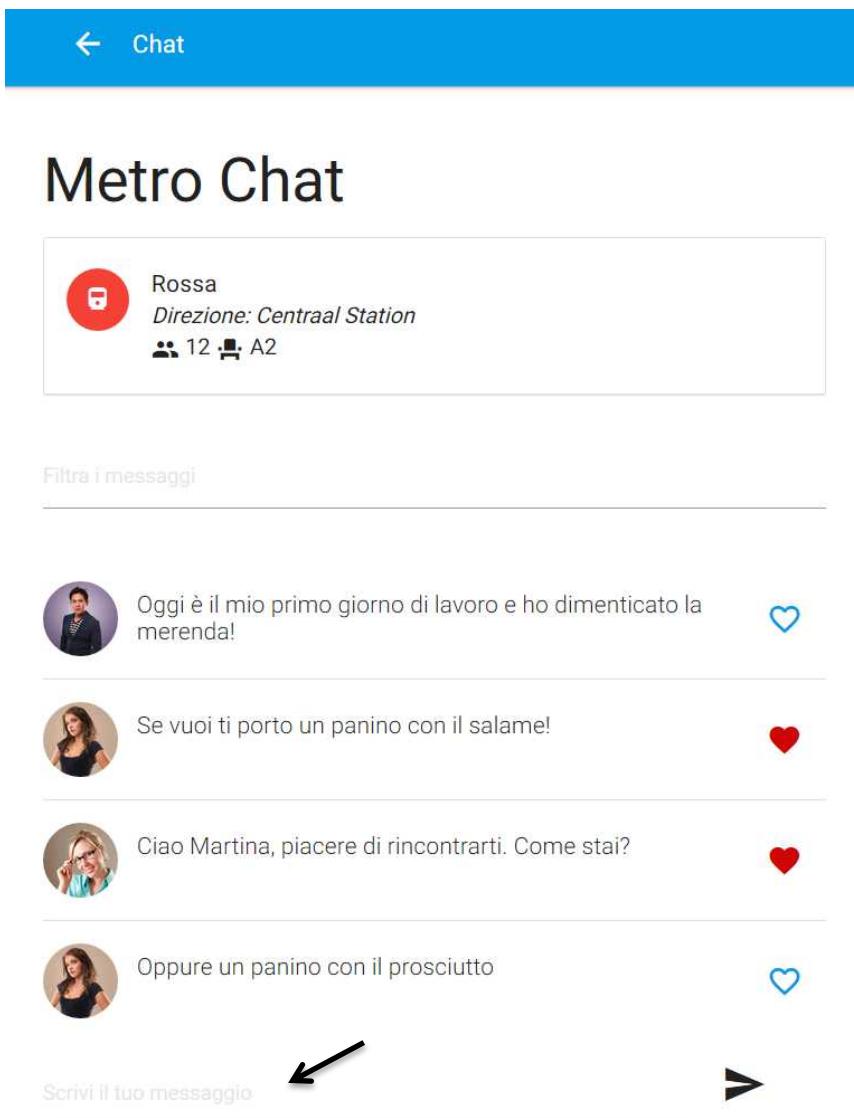


Figura 12.4 – Dettaglio messaggi scambiati nella linea Rossa con campo di input

E' chiaro che, per poter inviare il messaggio nella chat, dobbiamo avere un campo di input dove inserire il testo e sfruttando il metodo `post()`, inviarlo in remoto al fine di memorizzarlo nel database e renderlo visibile a tutti gli utenti collegati.

Le informazioni che dovremo memorizzare nell'ipotetico database remoto, dovranno essere:

- il testo del messaggio
- l'identificativo dell'utente che lo sta inviando
- l'identificativo del treno associato alla chat

Per ora ipotizzeremo che l'identificativo dell'utente sia noto, ma nella realtà dovremo recuperarlo dopo che l'utente ha effettuato il login nell'apposita sezione dell'applicazione.

Se nel server remoto fosse presente PHP, un possibile programma per il recupero delle informazioni passate in formato JSON potrebbe essere:

index.php

```
<?php
$json = file_get_contents('php://input');
$mypost = json_decode($json,true);

// recupero i dati passati da Angular
$idt = $mypost['idtreno'];
$messaggio = $mypost['messaggio'];

// TODO: salvo i dati nel database e poi...
// .....

// mostro una stringa nel formato JSON con l'id recuperato
header('Content-Type: application/json');
echo '{"dati":"' . $idt . '}';
```

La stringa restituita dal programma PHP è in formato JSON, quindi potrà essere sfruttata dall'applicazione Angular per mostrare un messaggio all'utente o un eventuale errore.

I dati che dovremo confezionare, sono indicati con le proprietà `idtreno`, poi `idutente` - per ora valorizzato manualmente al valore 99 - e `messaggio`.

```
const dati = {
    idtreno: 'AFG',
    idutente: '99',
    messaggio: 'Messaggio di test'
}
```

Siamo pronti per aggiungere la funzionalità d'invio dati a un nuovo Service, che gestirà tutta la sezione Chat dell'applicazione.

Possiamo ricopiare gran parte delle righe dell' equivalente servizio usato per mostrare la lista dei treni, aggiungendo anche il metodo `sendMsgChatObservable(obj)` creato con le nozioni viste fino ad ora.

Il Service lo potremmo chiamare `chat.service.ts`, e si occuperà sia di visualizzare l'elenco dei messaggi remoti, sia d'inviare il messaggio all'API creata in precedenza.

Dovremo quindi specificare i due differenti endpoint dell'API, come da indicazioni che troviamo nell'appendice del libro.

Il codice sarà:

```
service/chat.service.ts

//...
import { HttpClient, HttpHeaders } from '@angular/common/http';
import { throwError, Observable } from 'rxjs';
import { catchError, map, tap } from 'rxjs/operators';

@Injectable({providerIn: 'root'})

export class ChatService {
// 1
private apiUrl = 'https://www.dcopelli.it/test/angular/chat/';
private apiPostUrl = 'https://www.dcopelli.it/test/angular/chat/send/';
// 2
constructor(private http: HttpClient) {}

getListaChat(): Chat[] {
    // recupero i dati statici per ora
    return LISTAMSG;
}
// 3
getListaChatObservable(idt: string): Observable<Messaggio[]> {
    return this.http.get<Messaggio[]>(this.apiUrl + '?idt=' + idt)
        .pipe(
            map((risposta:any) => risposta['dati']),
            catchError(this.handleErrorObs));
}

// 4
sendChatMsgObservable(obj: Messaggio): Observable<Messaggio> {
    return this.http.post<Messaggio>(
        apiPostUrl,
        { idtreno: obj.idt,
            idutente: obj.idu,
            messaggio: obj.testo },
    )
} (continua)
```

```

        .pipe(
            map((risposta:any) => risposta['dati']),
            catchError(this.handleErrorObs));
    }

// 5
private handleErrorObs(error:any) {
    return throwError(error.message || error);
}

}

```

dove al posto dell'oggetto `Metro`, abbiamo sostituito l'oggetto `Messaggio`, il cui modello `messaggio.model.ts`, era stato progettato quando abbiamo parlato di modellazione dati.

Le parti interessanti sono quelle indicate con i punti //3 e //4.

```

// 3
getListaChatObservable(idt: string): Observable<Messaggio[]> {
    return this.http.get<Messaggio[]>(this.apiUrl + '?idt=' + idt)
        .pipe(
            map((risposta:any) => risposta['dati']),
            catchError(this.handleErrorObs));
}

//

```

L'interrogazione dell'API per il recupero dei messaggi presenti nella chat di un particolare treno, deve essere fatta inviando l'identificativo del treno e l'identificativo dell'utente.

Questo al fine di evidenziare eventuali messaggi inseriti dall'utente di prova – quello con `idu='99'` – tra i preferiti. Abbiamo costruito pertanto manualmente l'URL, concatenando alla stringa `apitGetUrl`, la stringa rappresentativa della query, per ottenere:

```
https://www.dcopelli.it/test/angular/chat/?idt=XXX&idu=99
```

Questa tecnica l'abbiamo già vista quando abbiamo recuperato i dati del treno. Una alternativa, più elegante, è quella di sfruttare la classe `HttpParams`, il metodo `set()` e la proprietà `params`.

La classe sarà importata sempre da `@angular/common/http` e sfrutteremo più chiamate al metodo `set()`, a seconda di quanti parametri query dobbiamo ipostare nell'URL.

```
const param = new HttpHeaders().set('idt', idt).set('idu', '99');
```

Il tutto dovrà poi essere inserito all'interno del parametro opzionale del metodo `get()`, sfruttando la proprietà `params`:

```
getListaChatObservable(idt: string): Observable<Messaggio[]> {
    return this.http.get<Messaggio[]>(
        this.apiGetUrl,
        { params: this.param }
    ).pipe(
        map((risposta:any) => risposta['dati']),
        catchError(this.handleErrorObs)
    );
}
```

Per quanto riguarda invece l'invio del messaggio, la cosa interessante da osservare è che il valore restituito da `sendChatMsgObservable()`, è un `Observable` di tipo `Messaggio`, che possiamo usare per recuperare l'identificativo del messaggio memorizzato nel server remoto e restituito dall'API.

Vediamo ora come usare il nuovo Service all'interno dell'applicazione. Potremmo creare una pagina dedicata alla lista dei messaggi chat, ma, avendo già sviluppato il componente di dettaglio, sfruttiamo quest'ultimo.

Come ricorderai, il codice sviluppato, si limitava a mostrare alcuni dati del treno, prelevati tramite il metodo `getDettaglioMetroObservable()` del Service `MetroService`.

```
treni/dettagliotreno/dettaglio.component.ts

//...
import { Metro } from './../../../../model/metro.model';
import { TreniService } from './../../../../service/treni.service';

@Component({
  selector: 'ca-dettaglio',
  template: `<div *ngIf="treno">
    <p> Treno Linea: {{treno.linea}} </p>
    <p> ID: {{treno.idt}}</p>
  </div>`
```

(continua)

```

})
export class DettaglioComponent implements OnInit {
  idtreno!: string;
  treno!: Metro;
  errormsg!: any;

  constructor(private route: ActivatedRoute,
              private treniservice:TreniService) {
}

ngOnInit() {
  this.idtreno = this.route.snapshot.paramMap.get('id')!;
  this.getDettaglioMetroObservable(this.idtreno);
}

getDettaglioMetroObservable(idt: string) {
  this.treniservice.getDettaglioMetroObservable(idt)
    .subscribe(
      (risp:any) => this.treno = risp[0],
      error => this.errormsg = error);
}

getDettaglioMetro(idtr: string) {
  this.treno = this.treniservice.getDettaglioMetro(idtr);
}
}

```

Quello che potremmo fare è aggiungere alla visualizzazione di queste informazioni, anche la lista di tutti i messaggi scambiati in chat dagli utenti del treno.

Questo ci permetterà di capire come si possano creare e gestire un certo numero di componenti, all'interno di uno stesso componente padre.

I messaggi restituiti dall'API potrebbero avere il formato indicato qui sotto (array di oggetti `Messaggio`), in cui sono presenti anche altre proprietà, oltre a quelle fino ad ora usate, che potrebbero essere sfruttate per ampliare le funzionalità dell'applicazione.

```

JSON Dati non elaborati Header
Salva Copia Comprimi tutto Espandi tutto Filtra JSON
▼ dati:
  ▼ 0:
    idm: "1"
    ▼ testo: "Oggi è il mio primo giorno di lavoro e ho"
    idu: "WRETY"
    idt: "12345"
    iddestinatario: ""
    stato: 1
  ▼ 1:
    idm: "2"
    testo: "Se vuoi ti porto un panino con il salame!"
    idu: "99999"
    idt: "12345"
    iddestinatario: "88888"
    stato: 1
  ▼ 2:
    idm: "22"
    testo: "Ciao Martina, piacere di rincontrarti. Cor"
    idu: "98"
    idt: "12345"
    iddestinatario: "ASD"
    stato: 1

```

Figura 12.5 – Rappresentazione JSON dei singoli messaggi scambiati nella linea Rossa

Ad esempio, la proprietà `idu`, dovrebbe far parte dell'applicazione non appena avrai inserito anche una sezione di login , al fine di identificare l'utilizzatore reale dell'applicazione, mentre `iddestinatario`, potrà essere usata nel caso tu vo glia inviare dei messaggi privati a quel particolare utente.

Anche la proprietà `stato` non è stata ancora inserita nel modello dei dati e ci servirà per contrassegnare un messaggio appena inserito nella lista dei preferiti di quel particolare utente.

Così come fatto per la lista dei treni, potremmo creare un componente dedicato per visualizzare la lista dei messaggi, quindi, un componente "intelligente" che riceva dei dati in ingresso.

index.html

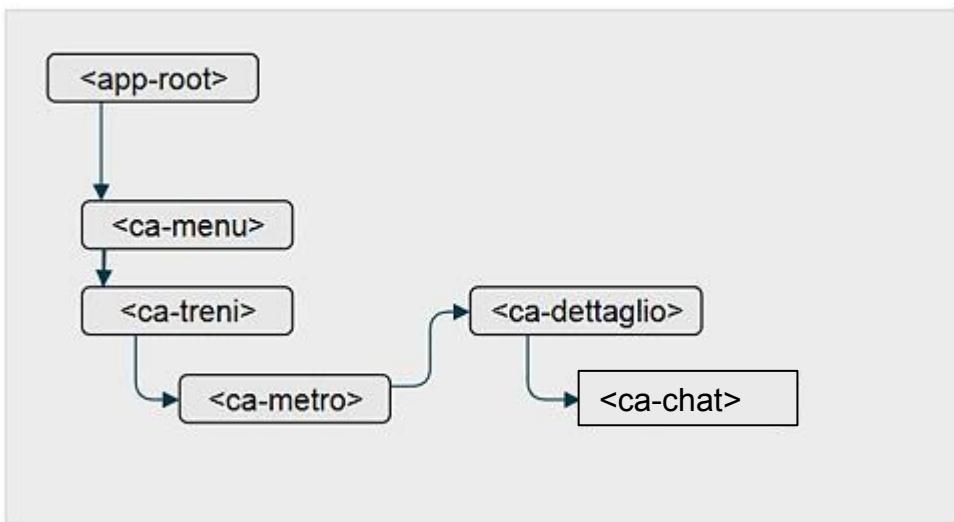


Figura 12.6 – Albero dei Componenti dell'app fino alla sezione dettaglio

A livello di selettore, potremmo inserirlo nel componente di dettaglio con una notazione di questo tipo:

```

template: `
  <div *ngIf="treno">
    <p> Treno Linea: {{treno.linea}} </p>
    <p> ID: {{treno.idt}}</p>
  </div>
  <ca-chat *ngFor="let chat of listachat" [msgIn]="chat"></ca-chat>
`
```

che ricorda quella vista per il componente *metro.component.ts*, dove *listachat*, sarà un array valorizzato con il metodo *getListaChatObservable(idt)* progettato per il service *chat.service.ts*

Il componente dettaglio pertanto diventa:

treni/dettagliotreno/dettaglio.component.ts

```

import { Component, OnInit } from '@angular/core';
import { ActivatedRoute, ParamMap } from '@angular/router';
import { Metro } from '../../../../../model/metro.model';
import { TreniService } from '../../../../../service/treni.service';
import { Messaggio } from '../../../../../model/messaggio.model';
  
```

(continua)

```

import { ChatService } from '../../../../../service/chat.service';

@Component({
  selector: 'ca-dettaglio',
  template: `
    <div *ngIf="treno">
      <p> Treno Linea: {{treno.linea}} </p>
      <p> ID: {{treno.idt}}</p>
    </div>
    <ca-chat *ngFor="let chat of listachat" [msgIn]="chat"></ca-chat>
  `
})

export class DettaglioComponent implements OnInit {
  idtreno!: string;
  treno!: Metro;
  errormsg!: any;
  listachat!: Messaggio[];

  // 1)
  constructor(
    private route: ActivatedRoute,
    private treniservice: TreniService,
    private chatservice: ChatService) {
  }

  ngOnInit() {
    this.idtreno = this.route.paramMap.get('id')!;
    this.getDettaglioMetroObservable(this.idtreno);
    // 2)
    this.getListChatObservable(this.idtreno);
  }

  getDettaglioMetroObservable(idt: string) {
    this.treniservice.getDettaglioMetroObservable(idt)
      .subscribe(
        (risp:any) => this.treno = risp[0],
        error => this.errormsg = error);
  }

  getListChatObservable(idt: string) {
    this.chatservice.getListChatObservable(idt)
      .subscribe(
        risp => this.listachat = risp,
        error => this.errormsg = error);
  }

  getDettaglioMetro(idtr:string) {
    this.treno = this.treniservice.getDettaglioMetro(idtr);
  }
}

```

L'unica osservazione da fare, è la presenza del parametro `idt` del metodo `getListChatObservable()` che corrisponde proprio all'id recuperato dall' URL dell'applicazione e trasmesso all'API al fine di selezionare solo i messaggi chat corrispondenti a quel treno.

Infine, il componente con selettore `<ca-chat>`, dovrà ricevere in ingresso un parametro, quindi lo progetteremo con le tecniche che già conosciamo. Lo possiamo salvare all'interno della cartella `chat`, con il classico nome `chat.component.ts`

```
chat/chat.component.ts

import { Component, OnInit, Input } from '@angular/core';
import { Messaggio } from './model/messaggio.model';

@Component({
  selector: 'ca-chat',
  template: `<p>{{msgIn.idu}}-{{msgIn.testo}} </p>`
})

export class ChatComponent implements OnInit {
  @Input() msgIn!: Messaggio;

  constructor() {}

  ngOnInit() {}
}
```

Il parametro d'ingresso è stato chiamato `msgIn` ed è di tipo `Messaggio`, in modo da poter visualizzare le singole proprietà come `idu` e `testo`. Ricordati di inserire l'operatore `!` per avvertire il compilatore che sarà inizializzato in seguito.

Il componente padre, potrebbe subire solo una piccola variazione nel testo mostrato all'utente, che invece di essere "Dettaglio Treno", diventerebbe "Chat Treno".

Eventualmente potresti anche cambiargli il nome, visto che `dettagliotreno.component.ts` potrebbe creare confusione.

Lascia a te questo come utile esercizio, perch è spesso vedrai ti capiterà di dover cambiare dei nomi ad un componente.

Bene, siamo riusciti a completare l'applicazione in modo che ora i dati siano visualizzati prelevandoli da una sorgente esterna. Ci rimane da vedere come dare la possibilità all'utente di inviare un nuovo messaggio, ma questo lo tratteremo quando parleremo delle basi dei moduli web.

12.4 Aggiornare dati in PUT

Oltre a POST e GET, Angular offre la possibilità di inviare i dati con intestazioni specifiche di modifica dati. La sintassi è sostanzialmente identica a quella di POST:

```
oggettoHttp.put(URL, DATI);
```

Nella forma base che useremo noi, il metodo restituisce un Observable di tipo Object che può essere usato con tutte le tecniche viste fino ad ora. I dati sono sempre interpretati nel formato JSON.

Un utile esercizio potrebbe essere quello di implementare un pulsante di "Aggiungi ai preferiti", in corrispondenza ai messaggi inviati in chat, in modo da salvare all'interno di una pagina specifica "preferiti", la lista delle persone che hai individuato per eventuali future chat.

In questo modo accedendo alla pagina "preferiti" potresti visualizzarli sotto forma di elenco, oltre che avere la possibilità di cancellarli, azione che impareremo a fare con il metodo `delete()`.

E' chiaro che ogni utente potrà avere la propria lista di preferiti, pertanto ipotizzando che il parametro `idutente` sia noto (es. 99), i dati che potresti memorizzare nel database remoto dopo il click sono, `idutente`, `idmessaggio` e lo stato.

```
{idutente: '99', idmessaggio: 'AAA', stato:1}
```

All'interno del service `chat.service.ts`, potresti aggiungere il metodo `setChatPreferiti()`, che riceve in ingresso i tre parametri da salvare, e restituisce un Observable di tipo `number`, che ci servirà per indicare se il messaggio dovrà apparire con a fianco un'icona selezionata, oppure deselectata, a seconda del valore restituito 1 o 0:

```
setChatPreferiti(idu: string,idm: string,stato: number): Observable<number> {
  return this.http.put<number>(this.apiPreferitiUrl,
    {idutente: idu, idmessaggio: idm, stato: stato}
  ).pipe(
    map((risposta:any) => risposta['stato']),
    catchError(this.handleErrorObs));
}
```

Chiaramente `apiPreferitiUrl` sarà valorizzata con l'url dell'API predisposta per effettuare il salvataggio dei dati (vedi appendice). Così facendo, all'interno del componente `<ca-chat>`, potrei aggiungere il pulsante per salvare il messaggio tra i preferiti, oppure un'icona che cambia stato e quindi colore, esattamente come fatto per la lampadina, quando abbiamo parlato di direttive.

Iniettando il Service nel costruttore e aggiungendo le relative righe di import, ottengo:

```

chat/chat.component.ts

import { Component, OnInit, Input } from '@angular/core';
import { Messaggio } from './model/messaggio.model';
import { ChatService } from './service/chat.service';

@Component({
  selector: 'ca-chat',
  template: `<p>{{msgIn.idu}} - {{msgIn.testo}}
    <span *ngIf="msgIn.stato==1; else show"
      (click)="setMsgPreferiti(msgIn.idm,0)">
      <i class="material-icons on">favorite</i>
    </span>
    <ng-template #show>
      <span (click)="setMsgPreferiti(msgIn.idm,1)">
        <i class="material-icons">favorite_border</i>
      </span>
    </ng-template>
  </p>
  <p>{{errormsg}}</p>
`)

export class MsgComponent implements OnInit {
  @Input() msgIn!: Messaggio;
  errormsg!: any;

  // Inietto il Service
  constructor(private chatService: ChatService) {}

  ngOnInit() {}

  setMsgPreferiti(idm:number, newstato:string) {
    // recupero lo stato attuale, per reimpostarlo in caso di errore
    const statoprec = this.msgIn.stato;
    this.msgIn.stato = newstato;
    this.chatService.setChatPreferiti('99', idm, newstato)
      .subscribe(
        risp => {if String(risp)!= newstato) {
          // risposta negativa del server
          alert("Errore");
          this.msgIn.stato = statoprec;
        }
      )
  }
}

```

(continua)

```

    } else {
        alert("Ok");
        this.msgIn.stato = newstato;
    },
error => this.errormsg = error);
}
}

```

dove nel template ho aggiunto un gestore di evento sul click di nome `setMsgPreferiti()` passandogli in ingresso l' `id` del messaggio da inserire tra i preferiti e lo stato (,,1'=aggiunta;'0'=eliminazione)

Il metodo del service, invierà questi dati in modalità PUT, insieme all'ipotetico identificativo dell'utente di test (stringa 99), al fine di memorizzarli nel database o di rimuoverli.

Per cambiare lo stato dell'icona associata ai preferiti, senza aspettare la risposta del server, ho valorizzato la proprietà `stato` dell'oggetto `msgIn`, con il valore passato al metodo e al ricevimento del dato effettivo, mostro un messaggio di `alert()` per verificare se la stato inviato dal server, coincide con quello impostato.

In caso di errore, reimposto il valore di `stato` a quello precedentemente memorizzato in `statoprec`, per evitare che l'utente veda un messaggio selezionato quando nel server non lo è, a seguito di un errore.

12.5 Cancellare i dati con DELETE

Altra operazione spesso necessaria è la cancellazione dei dati. Anche qui Angular offre un metodo specifico `delete()` che ha la stessa sintassi del metodo `get()`:

```
oggettoHttp.delete(URL)
```

Questa è la forma base, ma anche in questo caso esistono numerose firme, che prevedono l'inserimento di parametri opzionali.

Lascio a te fare delle sperimentazioni, ora che penso tu abbia capito come sfruttare le tecniche di recupero e invio informazioni.

Come dicevo in precedenza, un possibile sviluppo dell'app che stiamo progettando, è quello di aggiungere la sezione "Preferiti", che ci dia la possibilità di mostrare

l'elenco dei messaggi contrassegnati in precedenza, oltre che la possibilità di cancellarli dalla lista dei preferiti.

Lascio a te lo sviluppo come utile esercizio per capire se tutti i concetti spiegati fino ad ora sono stati correttamente assimilati.

Ti consiglio di partire con dei dati locali per poi passare ad una eventuale API remota.

Capitolo 13

Creazione e gestione di Moduli Web

13.1 I Form in Angular: introduzione

Come ogni sito web che si rispetti, anche un'applicazione Angular necessita di interagire con l'utente.

Questo avviene tipicamente o sotto forma di tocco /click, o sotto forma di dati da inserire in un campo di testo , ad esempio per effettuare delle ricerche o per trasmetterli ad un'API in grado di comunicare con un database remoto.

Nell'attesa che si possa interagire con le proprie applicazioni grazie a comandi vocali, vediamo come si possa aggiungere il classico modulo web all'interno di un componente, al fine di recuperare delle informazioni nec essarie ad interagire con l'applicazione.

Nel caso dell'app che stiamo progettando, l'unica sezione che richiede l'inserimento di dati da parte dell'utente, è quella d'invio del messaggio nella chat, quindi un solo campo di input che potrebbe essere gestito con una variabile locale nel template.



Ciao Martina, piacere di rincontrarti. Come
stai?



Scrivi il tuo messaggio



Figura 13.1 – Form per l'inserimento del messaggio da inviare nella chat del treno

La gestione dei form in Angular è molto flessibile, in quanto è possibile scegliere due strade completamente differenti, che dovranno essere valutate sulla base della complessità del modulo.

La prima strada che analizzeremo per la costruzione di questo modello, è quella chiamata "Template Driven", e va bene quando il modulo web è costituito da uno o due campi e non sono necessarie complesse azioni di verifica dati.

La seconda, chiamata "Reactive Form", va bene quando hai dei moduli web "complessi" ossia, non solo costituiti da diversi campi, ma con la necessità di verifica di coerenza dei dati basata su più fattori collegati.

Entrambe le strade offrono vantaggi e svantaggi, ma se provieni dallo sviluppo di siti web, forse la prima è quella che più si avvicina al tuo modo di lavorare. Con il tempo vedrai che la seconda è più immediata da applicare.

Nel corso del capitolo, vedremo tutte e due le tecniche, e amplieremo il discorso non solo ai campi di input, ma anche ai campi di select, checkbox, radiobox, al fine di riuscire a creare moduli web sempre più complessi.

Vediamo prima di tutto a cosa fanno riferimento le seguenti terminologie:

1. **Gestione Form "Template Driven"** – In questo caso, la logica di interazione con i diversi campi è gestita in gran parte nel template, così come avviene per le classiche pagine web
2. **Gestione Form "Reactive Form"** – In questo caso, la logica è gestita direttamente con proprietà interne al corpo della classe del componente.

13.2 Cosa verificare prima di usare i Form

Al fine di poter usare le funzionalità offerta da Angular legate ai moduli web, il primo passo da fare è andare a verificare/modificare il file `app.module.ts` presente nella cartella `app` del progetto, aggiungendo le righe per poter importare due moduli dal package `@angular/core`.

Nel caso volessi implementare il form usando il modello "Template Driven", al file `app.module.ts` dovrà aggiungere il modulo `FormsModule`.

Nel file `app.module.ts` dell'applicazione pertanto, dovrà scrivere:

app.module.ts

```

import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
// 1) riga per importare il Modulo FormModule
import { FormsModule } from '@angular/forms';
import { AppComponent } from './app.component';

@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
    // 2) rendo disponibile il modulo a tutta l'app
    FormsModule,
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }

```

Se hai creato il progetto sfruttando la linea di comando, queste righe potrebbero essere già state inserite per te. Parlo al condizionale, perché in alcune versioni di CLI potrebbe non accadere. Dovrai quindi verificare che ci siano, o procedere con l'inserimento nel caso mancassero.

Nel caso volessi invece sfruttare la tecnica “Reactive”, dovrà inserire un altro modulo, che prende il nome di `ReactiveFormsModule`.

app.module.ts

```

import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
// 1) riga per importare il Modulo FormModule
import { ReactiveFormsModule } from '@angular/forms';
import { AppComponent } from './app.component';

@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
    // 2) rendo disponibile il modulo a tutta l'app
    ReactiveFormsModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }

```

NB: Ricordati che l'operazione d'aggiunta di un nuovo modulo all'interno dell'array indicato dalla proprietà `import`, deve essere sempre accompagnata dalla riga di importazione della relativa classe.

13.3 Creare il Form con la tecnica “Template Driven”

Così come avviene per una classica pagina web, se dovessi progettare un semplice modulo web che richiede l'inserimento del nome e dell'email:

The form consists of two text input fields and a submit button. The first field is labeled "Nome" and has a placeholder "Il tuo nome". The second field is labeled "Email" and has a placeholder "La tua email". Below the fields is a green rectangular button with the text "INVIA" in white capital letters.

Figura 13.2 – Form d'esempio con campo Nome ed Email

a livello di codice HTML dovremmo scrivere del codice simile a questo:

```
<form action="" method="post">
  <label for="nome">Nome</label>
  <input type="text" name="nome" value="" placeholder="Il tuo Nome">
  <label for="email">Email</label>
  <input type="email" name="email" value="" placeholder="La tua Email">
  <button type="submit">Invia</button>
</form>
```

In Angular non è più necessario utilizzare l'attributo `action` e `method`, perché il tutto sarà gestito da funzionalità interne alla classe del componente. Analoghi discorsi per l'attributo `value` di ogni campo.

In Angular infatti, ogni modulo web, è gestito da una rappresentazione indipendente dall'interfaccia UI, chiamata "Form Model", che permette di modellare il form tramite tre classi fondamentali: `FormControl`, `FormGroup`, `FormArray`



Figura 13.3– Rappresentazione del modello alla base dei form in Angular

Grazie all'uso di opportune direttive, sarà in grado di collegare gli elementi del DOM con questo modello e gestire le operazioni di recupero delle informazioni così come la visualizzazione dei dati nel form.

La prima modifica allora che dovrai fare, rispetto alla classica progettazione di un modulo per una pagina web, sarà l'eliminazione degli attributi citati sopra:

```

<form>
  <label for="nome">Nome</label>
  <input type="text" name="nome" placeholder="Il tuo nome">

  <label for="email">Email</label>
  <input type="email" name="email" placeholder="La tua email">

  <button type="submit">Invia</button>
</form>

```

A questo punto dobbiamo capire come collegare gli elementi del DOM, con il modello usato da Angular per la gestione del form.

13.4 Gestire i dati da un campo <input>

Se nel mondo delle pagine web, il recupero dei dati da un modulo avviene inviandoli ad un'applicazione scritta in qualche linguaggio come PHP, Ruby, Python, ASP.net, NodeJS, nel caso di Angular devi dimenticarti tutto questo, perché i dati sono recuperati direttamente nel client, e poi eventualmente inviati all'applicazione remota.

Come già detto, non ci occuperemo di realizzare un'applicazione stile “WhatsApp”, ma di capire come avviene il processo d'invio dei dati dai campi di un modulo web.

NB: E' necessario prestare attenzione alla posizione in cui stai recuperando/visualizzando i dati del modulo, perché dovrà usare notazioni diverse a seconda tu stia lavorando sul template o sul corpo della classe.

Il form di partenza è quella visto in precedenza, a cui dovremo aggiungere un gestore di evento per l'azione di *submit* e un'opportuna sintassi per poter recuperare lo specifico campo inviato e mostrato, sia nel template che nel corpo della classe.

La prima cosa da fare è trasformare ogni campo del modulo in un oggetto "intelligente", in particolare in un oggetto ***FormControl***, sfruttando la direttiva `ngModel`, che è automaticamente disponibile all'interno di Angular, senza dover fare alcuna importazione.

La sintassi da usare per un generico campo input, sarà:

```
<input type="text" name="nome" ngModel>
```

o in modo del tutto equivalente:

```
<input type="text" name="nome" [ngModel]>
```

NB: Non devi assolutamente dimenticarti di inserire l'attributo *name* per ogni campo del form.

Vedremo che in realtà la direttiva offre la possibilità di **collegare** in entrambe le direzioni il campo, quindi non solo per leggerne il contenuto (dal template al componente), ma anche di impostarlo (dal componente al template).

Con la notazione indicata sopra, riusciamo a collegare una proprietà del template con il componente, ma non viceversa.

Per poter leggere e settare un dato, dovremo "incastrare" la direttiva all'interno della notazione detta **"banana in a box"**, perché i simboli usati, ossia le parentesi quadre e tonde, assomigliano a una scatola [] con all'interno due banane rappresentate dalle parentesi tonde ().

Tradotto in termini pratici, dovrà scrivere:

```
<input type="text" name="nomecampo" [(ngModel)]="nomecampo_modello">
```

La notazione in realtà fa riferimento a qualcosa che abbiamo già visto ossia il collegamento ad una proprietà (che viene fatto con le parentesi quadre), e la gestione

di un evento (che viene fatto con le parentesi tonde). Il tutto sotto il controllo di Angular e grazie a ngModel.

La riga precedente potrebbe essere infatti scritta così:

```
<input type="text" name="nomecampo"
       [value]="nomecampo_modello"
       (input)="nomecampo=$event.target.value">
```

Fai attenzione che nomecampo_modello è il riferimento usato nel corpo della classe per **assegnare** al campo un valore, mentre nomecampo, è il valore usato nel corpo della classe per **recuperare** il valore.

Nel template invece, si fa sempre riferimento a nomecampo_modello. Spesso si fa confusione perché vengono fatti coincidere, ma sono due soggetti diversi.

Fatte queste modifiche, dobbiamo procedere con l'ultimo passo, ossia capire come inviare queste informazioni, visto che nel modulo è stato tolto l'attributo action.

Per fare questo, si aggiunge al tag <form> un riferimento, rappresentato da una variabile locale con la classica notazione #variabile, valorizzata con ngForm e un gestore di evento ngSubmit, a cui passerò tale variabile.

Pertanto il modulo web precedente diventerà:

```
<form #mioform="ngForm" (ngSubmit)="invio(mioform)">
  <label for="nome">Nome</label>
  <input type="text" name="nome"
         placeholder="Il tuo nome"
         [(ngModel)]="nome_modello">
  <label for="email">Email</label>
  <input type="email" name="email"
         placeholder="La tua email"
         [(ngModel)]="email_modello">
  <button type="submit">Invia</button>
</form>
```

Le cose più interessanti da notare sono:

1. la **presenza in ogni campo** della direttiva ngModel che, come detto, permette di "potenziare" ogni campo del form trasformandolo in un oggetto

FormControl.

2. la presenza di una **variabile template** assegnata al tag <form>, chiamata a mio piacere #mioform, valorizzata con ngForm. In questo modo abbiamo un riferimento a tutti i campi "potenziati" del modulo.
3. la presenza della notazione (evento)="gestore" che abbiamo già visto quando abbiamo parlato di eventi in Angular. In questo caso l'evento è ngSubmit (già predefinito) e il gestore di tale evento è un metodo che dovrò progettare nel corpo della classe del componente. Osserva il nome del parametro di ingresso, che coincide proprio con quello scelto per la variabile template che referenzia il form.

In linea di massima, con queste tre semplici modifiche, sei in grado di trasformare un classico modulo web, adatto per un sito html, in un modulo web in grado di essere gestito in Angular.

Completiamo il codice, andando a creare un semplice componente di test, dal nome formiscriviti.component.ts, per capire come poter leggere i dati inseriti dall'utente nei due campi di input.

login/form-iscriviti.component.ts

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

@Component({
  selector: 'ca-form-iscriviti',
  template: `
    <form #mioform="ngForm" (ngSubmit)="invio(mioform)">

      <label for="nome">Nome</label>
      <input type="text" name="nome"
        placeholder="Il tuo nome"
        [(ngModel)]="nome_modello">

      <label for="email">Email</label>
      <input type="email" name="email"
        placeholder="La tua email"
        [(ngModel)]="email_modello">

      <button type="submit">Invia</button>
    </form>
    <p>Stai scrivendo nel campo nome: {{nome_modello}}</p>
    <p>Stai scrivendo nel campo email: {{email_modello}}</p>
  `

})

export class FormIscrivitiComponent {
  invio(form: NgForm) {
```

(continua)

```

    alert("Tutto il form: " + JSON.stringify(form.value));
    console.log("Nome inserito è " + form.controls['nome'].value);
    console.log("Email inserita è " + form.controls['email'].value);
}
}

```

Come puoi osservare, prima di tutto ho importato dal package `@angular/form, NgForm`, e poi ho creato il metodo `invio()`, che riceve in ingresso proprio il **riferimento alla variabile locale** del modulo, quindi un oggetto `ngForm`, che posso usare per recuperare i singoli campi, sfruttando la notazione:

```
oggetto.controls['nomecampo'].value
```

dove `nomecampo` è il nome dell' **attributo “name”** associato all'elemento del modulo nel DOM.

Quest'ultimo sarà proprio il nome del corrispondente oggetto `FormControl` creato da Angular, che potrai usare nel corpo della classe, per il recupero effettivo del dato inserito dall'utente.

Qui sotto puoi osservare come Angular vada a modificare l'elemento del DOM, con l'aggiunta non solo di `ng-reflect-name`, coincidente proprio con il nome assegnato all'attributo `name`, ma anche con l'aggiunta di una serie di attributi di classe, che potranno essere sfruttati per cambiare l'aspetto del campo, in funzione dello stato in cui si troverà.

```

▼ <form class="ng-untouched ng-pristine ng-valid"
novalidate=""> event
  <label for="nome">Nome</label>
  <input class="ng-untouched ng-pristine ng-valid"
name="nome" placeholder="Il tuo nome" type="text"
ng-reflect-name="nome"> event
▶ <label for="email"> ... </label>

```

Figura 13.4 – Attributi di classe inseriti da Angular in ogni elemento del form nel DOM

Nota anche la presenza nel template, di questi due paragrafi:

```
<p>Stai scrivendo nel campo nome: {{nome_modello}}</p>
<p>Stai scrivendo nel campo email: {{email_modello}}</p>
```

in cui abbiamo usato l'interpolazione per collegare le proprietà `nome_modello` e `email_modello`, assegnate a `ngModel`. Questo ci permette di accedere in tempo reale, al valore che l'utente sta inserendo nel campo input, e visualizzarlo così nel template.

Se provi a fare la stessa cosa in JavaScript, ti renderai conto della potenza offerta dalla direttiva `ngModel` e dalla notazione "banana-in-a-box"

Se avessimo usato la direttiva con notazione `[ngModel]`, quindi con un comportamento monodirezionale, non avremmo potuto valorizzare in tempo reale la proprietà collegata nel template.

E' possibile sfruttare l'oggetto `ngForm` anche per recuperare l'insieme di tutti i campi, accedendo alla proprietà `value`, come visto nella riga che mostra l'alert:

```
invio(form: NgForm) {
  alert("Tutto il form: " + JSON.stringify(form.value));
  //...
}
```

Una seconda tecnica, consiste nell'usare la variabile locale `#mioform`, che identifica il form, per accedere direttamente alla proprietà `value` di `ngForm`, che verrà passata come argomento al metodo `invio()`:

```
<form #mioform="ngForm" (ngSubmit)="invio(mioform.value)">
```

In questo modo possiamo accedere direttamente al valore associato all'oggetto `FormControl`, con la sintassi:

```
oggettoNgForm.nomecampo
```

dove `nomecampo`, coincide proprio con nome inserito all'interno dell'attributo `name` di ogni elemento del form.

L'unica cosa a cui devi fare attenzione è che il parametro passato al metodo `invio()`, ora non è più di tipo `NgForm`, ma sarà generico ossia `any`, in quanto passo direttamente il valore dei diversi campi.

Il codice del componente diventerebbe:

```
login/form-iscriviti.component.ts

import { Component } from '@angular/core';
import { NgForm } from '@angular/forms';

@Component({
  selector: 'ca-form-iscriviti',
  template: `
    <form #mioform="ngForm" (ngSubmit)="invio(mioform.value)">
      <label for="nome">Nome</label>
      <input type="text" name="nome"
             placeholder="Il tuo nome"
             [(ngModel)]="nome_modello">
      <label for="email">Email</label>
      <input type="email" name="email"
             placeholder="La tua email"
             [(ngModel)]="email_modello">
      <button type="submit">Invia</button>
    </form>

    <p>Stai scrivendo nel campo nome: {{nome_modello}}</p>
    <p>Stai scrivendo nel campo email: {{email_modello}}</p>
  `)

})

export class FormIscrivitiComponent {
  constructor() {}
  invio(form: any) {
    console.log("Il nome inserito è: " + form.nome);
    console.log("L'email inserita è: " + form.email);
  }
}
```

Un'altra proprietà dell'oggetto `ngForm` che è possibile sfruttare direttamente nel template del componente, è la proprietà `valid`, che ti permetterà di oscurare sezioni del form, in funzione della validità dei dati inseriti.

Ad esempio, potresti disabilitare il bottone d'invio, fino a che tutti i dati inseriti nei diversi campi non siano validi, come indicato qui sotto:

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

dove abbiamo reso obbligatori i campi di input con l'aggiunta di `required`:

```
login/form-iscriviti.component.ts

import { Component } from '@angular/core';
import { NgForm } from '@angular/forms';

@Component({
  selector: 'ca-form-iscriviti',
  template: `
    <form #mioform="ngForm" (ngSubmit)="invio(mioform.value)">

      <label for="nome">Nome</label>
      <input type="text" name="nome"
        placeholder="Il tuo nome"
        [(ngModel)]="nome_modello" required>

      <label for="email">Email</label>
      <input type="email" name="email"
        placeholder="La tua email"
        [(ngModel)]="email_modello"
        maxlength="64" required>

      <button type="submit" [disabled]="!mioform.form.valid">Invia</button>
    </form>
    <p>Stai scrivendo nel campo nome: {{nome_modello}}</p>
    <p>Stai scrivendo nel campo email: {{email_modello}}</p>
  `

})

export class FormIscrivitiComponent {
  invio(form:any) {
    console.log("Il nome inserito è: " + form.nome);
    console.log("L'email inserita è: " + form.email);
  }
}
```

Fino a che l'utente non ha inserito dei valori nei due campi, il pulsante di invio non sarà attivo.

13.5 Accedere ad un campo di `<input>` tramite una variabile nel template

Sulla base delle considerazioni viste fino ad ora, siamo pronti per inserire il campo di input che ci servirà per inviare il messaggio all'interno della chat scelta.

Hai chiaramente diverse possibili soluzioni, tra cui - la più semplice - quella di aggiungerlo direttamente all'interno del template del componente dettaglio.

In alternativa, potresti trasformarlo in un componente, nel caso avessi la necessità di usarlo in più sezioni dell'app.

Analizzando così il codice qui sotto associato al template del componente *dettaglio.component.ts* sviluppato in precedenza, puoi osservare una nuova notazione, che in realtà non è poi così nuova perché l'abbiamo già incontrata più volte:

```
<div *ngIf="treno">
  <p> Treno Linea: {{treno.linea}}</p>
  <p> ID: {{treno.idt}}</p>
</div>
<ca-chat *ngFor="let chat of listachat" [msgIn]="chat"></ca-chat>
<input type="text" #msg required>
<button (click)="invio(msg.value); msg.value=''">Invia</button>
```

Invece di usare `ngModel`, sfruttiamo una caratteristica dei template in Angular, che ci permette di usare una variabile locale contrassegnata dal simbolo `#`.

In questo modo, limitandoci ad usarla solo nel template, puoi accedere al valore inserito dall'utente nel campo di input associato, e inviarlo come parametro di ingresso ad un metodo definito nella classe.

```
<button (click)="invio(msg.value); msg.value=''">Invia</button>
```

L'evento che monitoriamo è il `click` e nota come abbia inserito un'espressione costituita dal richiamo del metodo `invio()` e dalla valorizzazione della proprietà `value` del campo input identificato da `msg`.

Queste due istruzioni, vengono eseguite in successione, avendo così la possibilità di cancellare la stringa inserita dall'utente, subito dopo che il dato è stato inviato.

Una volta ricevuto il dato, cosa dobbiamo fare? Dovremo visualizzarlo, quindi aggiungerlo all'array attualmente presente nella pagina e contemporaneamente inviarlo a server remoto al fine di memorizzarlo nel database.

Per quest'ultima operazione dovranno avvalerci del Service creato in precedenza, a cui aggiungeremo un metodo per l'invio dei dati in POST.

Una bozza di codice potrebbe essere:

treni/dettagliotreno/dettaglio.component.ts

```
//...
@Component({
  selector: 'ca-dettaglio',
  template: `<div *ngIf="treno">
    <p>Chat Treno: {{treno.linea}}</p>
    <p> ID: {{treno.idt}}</p></div>
    <ca-chat *ngFor="let chat of listachat" [msgIn]="chat"></ca-chat>
    <input type="text" #msg required>
    <button (click)="invioMsg(msg.value);msg.value=''">Invia</button>
  `
})
export class DettaglioComponent implements OnInit {
  idtreno!: string;
  treno!: Metro;
  listachat!: Messaggio[];
  chatmsg!: Messaggio;
  errormsg!: any;
  constructor(private route: ActivatedRoute,
              private treniservice:TreniService,
              private chatservice:ChatService) {
  }

  ngOnInit() {
    this.idtreno = this.route.snapshot.paramMap.get('id')!;
    this.getDettaglioMetroObservable(this.idtreno);
    this.getListChatObservable(this.idtreno);
  }
}

// 1) Costruisco l'oggetto Messaggio con i dati noti
invioMsg(testomsg: string) {
  // utente di test generico pari a 99 e idm impostato a 0
  this.chatmsg = {'idm': 0,
                  'idt': this.idtreno,
                  'idu': '99',
                  'testo': testomsg};
  this.sendMsgObservable(this.chatmsg);
}

// 2) Attendo che l'API risponda con oggetto Messaggio
sendMsgObservable(msg: Messaggio) {
  this.chatservice.sendMsgChatObservable(msg)
    .subscribe(
      (risp:any) => this.listachat.push(risp[0]),
      error=>this.errormsg = error);
}

getListChatObservable(idt: string) {
  this.chatservice.getListChatObservable(idt)
    .subscribe(
      risp => this.listachat = risp,
      error=>this.errormsg = error);
}
```

(continua)

```

getDettaglioMetroObservable(idt: string) {
    this.treniservice.getDettaglioMetroObservable(idt)
        .subscribe(
            (risp:any) => this.treno = risp[0],
            error=>this.errormsg = error);
}
}

```

Il codice si commenta da solo, nel senso che abbiamo confezionato l'oggetto `chatmsg`, rispettivamente con i dati del treno, dell'ipotetico utente, il messaggio, e un `idm` impostato a 0 perch é ancora non sappiamo il reale id entificativo che verrà assegnato dopo il salvataggio in remoto.

Questo oggetto è stato poi passato al metodo interno `sendMsgObservable()`, che attenderà la risposta del relativo metodo impostato nel `chat.service.ts`. La risposta dell'API, sarà un array costituito da un singolo oggetto, ed è questo il motivo per cui dovrai accedere al primo elemento dell'array, aggiungendolo all'attuale lista.

```
this.listachat.push(risp[0])
```

Chiaramente questa è una possibile soluzione, ma avendo il controllo dell'API remota, posso decidere di adottare questa strategia. Le cose cambierebbero se non avessi questo controllo, dovendomi adattare a quello impostato da altri.

Nel service `chat.service.ts` invece, dovrai creare il metodo che effettuerà l'effettivo invio al server remoto.

Il metodo lo avevamo già creato in precedenza, e gli passeremo un oggetto `Messaggio` con le diverse proprietà già valorizzate, al fine di ricostruire i parametri da passare e creare una stringa JSON:

```

sendMsgChatObservable(obj): Observable<Messaggio> {

    return this.http.post<Messaggio>(this.apiPostUrl, {
        idtreno: obj.idt,
        idutente: obj.idu,
        messaggio: obj.testo})
        .pipe(
            map((risp:any) => risp['dati']),
            catchError(this.handleError)
        );
}

```

L'API remota, sarà progettata al fine di ricevere i parametri `idtreno`, `idutente`, `messaggio` e di restituire un oggetto messaggio comprensivo anche del parametro `idm` valorizzato al valore effettivo e non più zero.

Questo è una delle possibili soluzioni, che presenta lo vantaggio che è necessario attendere la risposta dell'API prima di vedere il messaggio apparire nella lista.

Avresti potuto aggiungere subito il messaggio all'array e poi effettuare l'invio, ma dovresti poi cambiare la logica, nel caso di errore, in modo da cancellare il messaggio o segnalare all'utente che non è stato memorizzato.

Esercitazione facoltativa: potresti provare a progettare la sezione di login dell'applicazione MetroChat, aggiungendo il componente di nome `login.component.ts` con due campi di input: `username` e `password`.

Inoltre potresti disabilitare il bottone non appena l'utente clicca sul pulsante di login, e abilitarlo alla ricezione di una risposta dell'API remota.

13.6 Settare un dato all'interno di un campo di <input>

Ora che abbiamo visto le diverse tecniche per recuperare un dato, vediamo come si possa aggiornare un valore all'interno di un campo di input.

Questa operazione è richiesta ad esempio tutte le volte in cui vuoi recuperare dei dati precedentemente memorizzati in una sorgente esterna al fine di modificarli.

Chiaramente l'operazione dovrà essere fatta nel corpo della classe . Dovrai pertanto definire una proprietà, con lo stesso nome assegnato alla direttiva `ngModel`, e valorizzarla con un valore.

La sintassi da usare sarà:

```
nomecampo_modello = "valore da impostare"
```

dove `nomecampo_modello` è proprio una proprietà della classe , collegata grazie a `ngModel`, al relativo campo presente nel template.

Prendendo l'esempio iniziale, se volessi assegnare al campo nome il mio nome "Davide" dovresti scrivere:

login/form-iscriviti.component.ts

```

import { Component, OnInit } from '@angular/core';
import { NgForm } from '@angular/forms';

@Component({
  selector: 'ca-form-iscriviti',
  template: `
    <form #mioform="ngForm" (ngSubmit)="invio(mioform.value)">
      <label for="nome">Nome</label>
      <input type="text" name="nome"
        placeholder="Il tuo nome" [(ngModel)]="nome_modello">

      <label for="email">Email</label>
      <input type="email" name="email"
        placeholder="La tua email" [(ngModel)]="email_modello">

      <button type="submit">Invia</button>
    </form>
    <p>Stai scrivendo nel campo nome: {{nome_modello}}</p>
    <p>Stai scrivendo nel campo email: {{email_modello}}</p>
  `

})
export class FormIscrivitiComponent implements OnInit {

  nome_modello:string = '';
  constructor() {}
  ngOnInit() {
    this.nome_modello = "Davide";
  }
}

```

dove ho definito una proprietà nome_modello, valorizzata all'interno di ngOnInit()

13.7 Gestire i dati di un campo di <select>

I campi di select, sono costituiti da un elenco di valori selezionabili tramite il classico menu a tendina, quindi nell'ipotesi tu voglia ricreare in Angular l'equivalente tag HTML rappresentativo di alcune province italiane, come quello qui sotto, sono sicuro ti venga in mente di sfruttare la direttiva *ngFor::

```

<select name="pv">
  <option value="MI">Milano</option>
  <option value="BO">Bologna</option>
  <option value="NA">Napoli</option>
  <option value="CT">Catania</option>
</select>

```

Infatti se avessimo un array valorizzato in questo modo:

```
prov = [
    {valore: 'NA', nome: 'Napoli'},
    {valore: 'BO', nome: 'Bologna'},
    {valore: 'MI', nome: 'Milano'},
    {valore: 'CT', nome: 'Catania'}
];
```

nel template potremmo sfruttare la direttiva `*ngFor` per mostrare l'elenco:

```
<select name="pv">
    <option *ngFor="let pv of prov" value="{{pv.valore}}">
        {{ pv.nome }}
    </option>
</select>
```

Analogamente a quello visto per il campo `input`, vediamo come si possa sfruttare `ngModel` per recuperare il valore scelto dall'utente. Costruiamo allora il modulo, aggiungendo `ngModel` in corrispondenza al campo di `select`.

```
<form #mioform="ngForm" (ngSubmit)="invio(mioform)">
    <label for="pv">Provincia</label>
    <select name="pv" [(ngModel)]="pv_modello">
        <option *ngFor="let pv of prov" value="{{pv.valore}}">
            {{ pv.nome }}
        </option>
    </select>
    <button type="submit">Invia</button>
</form>
<p>Provincia selezionata {{pv_modello}}</p>
```

Come vedi ho aggiunto la classica notazione "banana-in-a-box", inserendo il nome `pv_modello` per distinguerlo dall'attributo `name`.

Avendolo inserito all'interno delle parentesi quadre e tonde, significa che posso sia leggerne il valore che mostrarlo nel template in tempo reale, non appena cambia.

All'interno del corpo della classe invece, dovrà sfruttare la stessa identica notazione vista in precedenza per il campo di `input`, quindi:

```
form.controls['pv'].value
```

o in alternativa, se passassi direttamente `mioform.value` al metodo invio , nella forma:

```
form.pv
```

dove ti ricordo `pv` è il nome impostato per l'attributo `name` del campo. Il codice completo diventa:

login/form-iscriviti.component.ts

```
//...
@Component({
  selector: 'ca-form-iscriviti',
  template: `
<form #mioform="ngForm" (ngSubmit)="invio(mioform.value)">
  <label for="pv">Provincia</label>
  <select name="pv" [(ngModel)]="pv_modello">
    <option *ngFor="let pv of prov" value="{{pv.valore}}">
      {{ pv.nome }}
    </option>
  </select>
  <button type="submit">Invia</button>
</form>
<p>Provincia selezionata {{pv_modello}}</p>
`)

export class FormIscrivitiComponent {
  constructor() {}
  invio(form: any) {
    alert("Provincia selezionata: " + form.pv);
  }
}
```

13.8 Settare un valore all'interno di un campo <select>

Così come visto per il campo di input, i potizzando che nel database remoto, per la provincia sia stato selezionato dall'utente il valore corrispondente a Bologna, come faccio a settarlo all'interno del campo di select, in modo che sia visualizzato selezionato all'apertura del componente?

Dobbiamo sfruttare il collegamento tra il campo select de l template e la propri età `pv_modello`, ottenuta grazie alla direttiva `ngModel`

Impostando pertanto il valore alla stringa “BO”, ottengo:

```
this.pv_modello="BO"
```

In questo modo, tra la lista dei valori presenti nel campo di select, sarà visualizzato quello corrispondente a Bologna.

login/form-iscriviti.component.ts

```
//...
@Component({
  selector: 'ca-form-iscriviti',
  template: `
    <form #mioform="ngForm" (ngSubmit)="invio(mioform.value)">
      <label for="pv">Provincia</label>
      <select name="pv" [(ngModel)]="pv_modello">
        <option *ngFor="let pv of prov" value="{{pv.valore}}">
          {{ pv.nome }}
        </option>
      </select>
      <button type="submit">Invia</button>
    </form>
    <p>Provincia selezionata {{pv_modello}}</p>
  `)
export class FormIscrivitiComponent {
  pv_modello: string;
  constructor() {
    this.pv_modello = 'BO';
  }
  invio(form: any) {
    alert("Provincia selezionata: " + form.pv);
  }
}
```

13.9 Gestire i dati di un campo <checkbox>

Il campo checkbox e il campo radio, come ben saprai, possono assumere due stati: selezionato o non selezionato. Se ad esempio volessimo dare la possibilità all'utente di selezionare delle categorie di interesse, dovremmo inserire un campo checkbox:

```
<input type="checkbox" name="catA">
<input type="checkbox" name="catB">
```

Come sempre in Angular è necessario trasformarlo in un `FormControl`, grazie alla direttiva `ngModel`, quindi la prima modifica da fare sarà aggiungerla per ognuno. Ipotizzando d'inserire sempre il collegamento bidirezionale per vedere gli effetti in tempo reale nel template, scriveremo:

```
<input type="checkbox" name="catA" [(ngModel)]="catA_modello">
<input type="checkbox" name="catB" [(ngModel)]="catB_modello">
```

dove, come già detto più volte, abbiamo usato delle stringhe diverse per l'attributo `name`, al fine di capire i diversi ruoli. Aggiungiamo questi due campi nel template:

```
<form #mioform="ngForm" (ngSubmit)="invio(mioform.value)">
  <label for="catA">Categoria A</label>
  <input type="checkbox" name="catA" [(ngModel)]="catA_modello">
  <label for="catB">Categoria B</label>
  <input type="checkbox" name="catB" [(ngModel)]="catB_modello">
  <button type="submit">Invia</button>
</form>
<p>Categoria A: {{catA_modello}}</p>
<p>Categoria B: {{catB_modello}}</p>
```

Il risultato che otteniamo è che al cambiamento dello stato di uno qualsiasi dei campi, sarà visualizzata la stringa "true" o "false", dove con "true" s'intende un campo selezionato.

Indipendentemente dallo stato, il recupero del valore "true" o "false" all'interno della classe, avviene con la seguente notazione:

```
this.campo_modello
```

dove con `campo_modello`, intendiamo il nome scelto per valorizzare `ngModel`.

Il codice completo diventa:

```
//...
@Component({
  selector: 'ca-form-iscriviti',
  template:
    <form #mioform="ngForm" (ngSubmit)="invio(mioform.value)">
      <label for="catA">Categoria A</label>
      <input type="checkbox" name="catA" [(ngModel)]="catA_modello">
      <label for="catB">Categoria B</label>
      <input type="checkbox" name="catB" [(ngModel)]="catB_modello">
      <button type="submit">Invia</button>
    </form>
    <p>Categoria A: {{catA_modello}}</p>
    <p>Categoria B: {{catB_modello}}</p>
```

(continua)

```

<input type="checkbox" name="catA" [(ngModel)]="catA_modello">
<label for="catB">Categoria B</label>
<input type="checkbox" name="catB" [(ngModel)]="catB_modello">
<button type="submit">Invia</button>
</form>
<p>Categoria A: {{catA_modello}}</p>
<p>Categoria B: {{catB_modello}}</p>
`

})
export class FormIscrivitiComponent {
  catA_modello:boolean = false;
  catB_modello:boolean = false;
  constructor() {}
  invio(form: any) {
    alert("Valore checkbox: " + this.catA_modello);
  }
}

```

Nel caso invece volessi impostare un valore, trattandosi di valori booleani, dovremmo impostare una proprietà con lo stesso nome collegato a ngModel e valorizzarla con true o false.

Ad esempio, se volessimo impostare la categoria B a llo stato “selezionata”, dovremmo scrivere:

```

login/form-iscriviti.component.ts

//...
@Component({
  selector: 'form-iscriviti',
  template: `
<form #mioform="ngForm" (ngSubmit)="invio(mioform.value)">
  <label for="catA">Categoria A</label>
  <input type="checkbox" name="catA" [(ngModel)]="catA_modello">
  <label for="catB">Categoria B</label>
  <input type="checkbox" name="catB" [(ngModel)]="catB_modello">
  <button type="submit">Invia</button>
</form>
<p>Categoria A: {{catA_modello}}</p>
<p>Categoria B: {{catB_modello}}</p>
`)

})
export class FormIscrivitiComponent {
  catA_modello:boolean = false;
  catB_modello:boolean = false;
  constructor() {
    this.catB_modello = true;
  }
  invio(form: any) {alert("Valore checkbox" + this.catA_modello);}
}

```

13.10 Gestire i dati di un campo <radio>

Come dicevamo in precedenza, questo campo può assumere due stati. Ad esempio se avessi bisogno dell' informazione sul sesso di una persona (Maschio o Femmina), potresti memorizzati i dati in un array, così come fatto per il campo checkbox:

```
sesso = [
    {valore: 'M', nome: 'Maschile'},
    {valore: 'F', nome: 'Femminile'}
];
```

e sfruttare la direttiva `*ngFor`, per creare i due campi di input, con collegata la direttiva `ngModel`, con le stesse tecniche viste per i campi precedenti:

```
<div *ngFor="let genere of sesso">
    <input type="radio" name="sesso"
        value="{{genere.valore}}"
        [(ngModel)]="sesso_modello"> {{genere.nome}}
</div>
```

Il recupero del valore, avviene con le stesse tecniche viste per il campo checkbox, quindi scriveremo:

```
login/form-iscriviti.component.ts

//...
@Component({
  selector: 'ca-form-iscriviti',
  template: `
    <form #mioform="ngForm" (ngSubmit)="invio(mioform.value)">
      <label for="cat">Maschio o Femmina</label>
      <div *ngFor="let genere of sesso">
        <input type="radio" name="sesso"
            value="{{genere.valore}}"
            [(ngModel)]="sesso_modello"> {{genere.nome}}
      </div>
      <button type="submit">Invia</button>
    </form>
    <p>Sesso: {{sesso_modello}}</p>
  `
})
export class FormIscrivitiComponent {
  constructor() {}
  invio(form: any) {
    alert("Valore radiobox: " + this.sesso_modello);
  }
}
```

Per poter impostare un valore nel campo invece, dovrà ai valorizzare una proprietà avente lo stesso nome di quello associato al campo tramite `ngModel`.

Ad esempio se volessi mostrare un valore predefinito pari a “M”, dovresti scrivere:

```
this.sesso_modello = 'M';
```

Il codice completa diventa:

```
//...
@Component({
  selector: 'ca-form-iscriviti',
  template: `
<form #mioform="ngForm" (ngSubmit)="invio(mioform.value)">
  <label for="cat">Maschio o Femmina</label>
  <div *ngFor="let genere of sesso">
    <input type="radio" name="sesso"
           value="{{genere.valore}}"
           [(ngModel)]="sesso_modello"> {{genere.nome}}
  </div>
  <button type="submit">Invia</button>
</form>
<p>Sesso: {{sesso_modello}}</p>
`)

export class FormIscrivitiComponent {
  sesso_modello:string;
  constructor() {
    this.sesso_modello = 'F';
  }
  invio(form: any){alert("Valore radiobox: " + this.sesso_modello);}
}
```

13.11 Creare il Form con la tecnica “Reactive”

Sebbene la nostra applicazione non preveda la costruzione di un form complesso, è utile approfondire anche l’alternativa “Reactive” fornita da Angular.

Come dicevamo all’inizio si tratta di una tecnica diversa dalla precedente in quanto l’attenzione non si focalizza più sul template, ma ci si sposterà nel corpo della classe dove si sfrutteranno una serie di classi che permettono la creazione dinamica del form e di gestirlo.

La cosa importante da ricordare è che per poter sfruttare le successive tecniche è necessario importare nel modulo radice il modulo `ReactiveFormsModule` come ricordato all'inizio del capitolo.

Le classi che tipicamente si usano per rappresentare il Modello Form, dicevamo all'inizio sono `FormBuilder`, `FormControl`, e `FormGroup` prelevabili dalla libreria `@angular/forms`.

Ad esempio la prima, come dice lo stesso nome, permette di costruire il form, un po' come se stessimo scrivendo il tag `form` nel template. La seconda invece permette di creare i singoli campi del form, quindi i campi `input`, `checkbox`, `radio`, `select`, mentre l'ultima permette di associare più campi del form sotto la stessa "casa", utile quando sia hanno più form da gestire nella stessa vista.

Vediamo allora da dove si parte nella costruzione di un form a livello di codice. Useremo gli stessi esempi visti nella prima parte in modo da avere una corrispondenza tra i due metodi.

Anche per i form "Reactive", la base di partenza è il vecchio classico modulo web HTML definito nel vista del componente .

Useremo pertanto lo stesso codice visto nel paragrafo 13.3 eliminando qualche altro attributo non necessario come `name`.

```
<form>
  <label for="nome">Nome</label>
  <input type="text" placeholder="Il tuo nome">
  <label for="email">Email</label>
  <input type="email" placeholder="La tua email">
  <button type="submit">Invia</button>
</form>
```

Il secondo passo è quello di aggiungere delle nuove direttive: `formGroup` e `formControlName`. I nomi sono abbastanza eloquenti. La prima ci permetterà di collegare il form presente nella vista, con un'istanza della classe `FormGroup` mentre la seconda con un'istanza della classe `FormControl`.

Chiaramente la prima andrà applicata all'intero form, mentre la seconda ai singoli campi. In questo modo creiamo un collegamento tra l'elemento del DOM e il nostro modello.

La sorgente dei dati sarà

login/form-iscriviti.component.ts

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

@Component({
  selector: 'ca-form-iscriviti',
  template: `
    <form [formGroup]= "mioForm">
      <label for="nome">Nome</label>
      <input type="text" placeholder="Il tuo Nome"
             formControlName="nome">
      <label for="email">Email</label>
      <input type="email" placeholder="La tua email"
             formControlName="email">
      <button type="submit">Invia</button>
    </form>`})
export class FormIscrivitiComponent {}
```

Come vedi tra virgolette abbiamo inserito i nomi delle variabili che andremo a sfruttare e a inizializzare nel corpo della classe del componente. Chiaramente dovrà scegliere dei nomi che ti permettano di ricordarne il significato dei diversi campi e il tipo di modulo.

Nella stessa pagina infatti potremmo avere più moduli e quindi più direttive `FormGroup` con relativi nomi identificativi.

Come sempre, essendo delle nuove classi, dovranno essere importate dalla relativa libreria `@angular/forms`.

Se ora provassimo a compilare il programma, sarebbe segnalato l'errore seguente:

 ▶ **ERROR Error: "formGroup expects a FormGroup instance. Please pass one in.**

Example:

```
<div [formGroup]="myGroup">
  <input formControlName="firstName">
```

Figura 13.5 – Errore segnalato in assenza dell'istanza `FormGroup` nel corpo della classe

Questo perché non abbiamo ancora creato l'istanza di `FormGroup` e Angular non è in grado di farlo da solo.

Dobbiamo manualmente creare ogni elemento del form a livello di codice TypeScript, quindi ripetere – questa volta con pure codice – le stesse righe inserite nel template.

Tutte le operazioni che faremo sulle diverse istanze che creeremo, si rifletteranno anche sulla vista, e saranno visualizzate dall’utente grazie alle direttive inserite, che creano il collegamento tra gli elementi del form e le diverse istanze presenti.

Il metodo più usato per generare tale codice è quello di sfruttare la classe `formBuilder` (anche questa da importare e iniettare nel costruttore) che permette la scrittura di codice meno prolioso.

Creeremo dunque l’istanza di `FormGroup` con un solo passaggio, grazie al metodo `group()` della classe `FormBuilder`, e la assegneremo alla variabile `mioForm`, proprio il nome usato nel template in corrispondenza alla direttiva `formGroup`:

login/form-iscriviti.component.ts

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

@Component({
  selector: 'ca-form-iscriviti',
  template: `
    <form [formGroup]= "mioForm">
      <label for="nome">Nome</label>
      <input type="text" placeholder="Il tuo nome"
             formControlName="nome">
      <label for="email">Email</label>
      <input type="email" placeholder="La tua email"
             formControlName="email">
      <button type="submit">Invia</button>
    </form>`
})
export class FormIscrivitiComponent {
  private mioForm: FormGroup;

  constructor(private fb: FormBuilder) {
    ngOnInit() {
      this.mioForm = this.fb.group({
        nome: '',
        email: ''
      })
    }
  }
}
```

Abbiamo iniettato nel costruttore la classe `FormBuilder`. L’oggetto JavaScript `group()`, ci permette di creare e

inizializzare le diverse istanze dei campi del modulo. In pratica è come se avessimo scritto tante righe di questo tipo:

```
nomecampo = new FormControl('');
```

A ciascuna istanza, inoltre, abbiamo assegnato un valore iniziale, che sarà quello mostrato non appena il form è visualizzato nel browser.

Ispezionando il codice HTML della pagina in corrispondenza all'elemento form, troveremo:

```
▼ <form class="ng-untouched ng-pristine ng-valid" novalidate="" ng-reflect-form="[object Object]"> event
  <label for="nome">Nome</label>
  <input class="ng-untouched ng-pristine ng-valid" formcontrolname="nome" placeholder="Il tuo nome" type="text" ng-reflect-name="nome"> event
  ▶ <label for="email">...</label>
```

Figura 13.6 – Attributi di classe inseriti da Angular in ogni elemento del form nel DOM

Come vedi ritroviamo gli stessi attributi di classe visti in precedenza (fig. 13.2) ad indicare che le due tecniche di costruzione dei form, producono lo stesso risultato.

Nota anche l'aggiunta degli attributi di classe `ng-untouched`, `ng-pristine`, `ng-valid`, che possono essere sfruttati per cambiare lo stile del campo del form in funzione di controlli di validità, come vedremo in uno dei prossimi paragrafi.

Arrivati a questo punto possiamo iniziare a spiegare come operare sui singoli campi, sia per le operazioni di recupero che di visualizzazione di nuovi dati.

13.12 Gestire i dati da un campo <input>

Ora che tutti i campi del modulo sono stati trasformati in un oggetto `FormControl`, possiamo comunicare in modo bidirezionale tra la vista e la classe, e tra la classe e la vista, sfruttando non più la notazione “banana in a box” ma direttamente l’istanza di `FormGroup` con cui ho accesso a tutti gli elementi del form.

Sarà infatti sufficiente usare il metodo `get()` per recuperare in tempo reale i dati inseriti in ogni campo del modulo.

Tale metodo restituisce un oggetto di tipo `AbstractControl`, classe che ha molte proprietà che permettono di recuperare informazioni importanti su ogni controllo. Una tra queste è la proprietà `value`:

```
nomeform.get('nomecampoform').value → snapshot
```

Tramite la proprietà `value`, ottengo un “fermo immagine” o `snapshot`, come si dice in gergo, del valore inserito dall’utente nel campo, mentre sfruttando la proprietà `valueChanges`:

```
nomeform.get('nomecampoform').valueChanges → Observable
```

ottengo un `Observable` a cui posso iscrivermi per recuperare il flusso di pressione dei tasti, quindi il flusso di lettere, di volta in volta inserite dall’utente (capiremo a fondo il concetto di `Observable` nel capitolo 14).

Se infatti inizializziamo con la stringa “Davide” solo il campo nome e all’interno di `ngOnInit` inseriamo le righe per recuperare il valore:

```
login/form-iscriviti.component.ts

//...
constructor(private fb: FormBuilder) {
    this.mioForm = this.fb.group({
        nome: 'Davide',
        email: ''
    });
}
ngOnInit() {
    console.log("Nome inserito è " + this.mioForm.get('nome').value);
    this.mioForm.get('nome').valueChanges
        .subscribe(tasto =>
            console.log("Tasto premuto è: " + tasto)
        )
}
```

all’apertura della pagina nel browser, nella console visualizzeremo subito il valore predefinito “Davide”, ma nient’altro.



Figura 13.7 – Snapshot del dato inserito nel campo nome

Questo perché l'Observable generato da `valueChanges` inizia ad emettere dei valori, solo dopo aver digitato qualcosa nel campo. Per ogni lettera digitata, sarà emesso il messaggio. Difatti non appena digitiamo uno spazio e le prime lettere del mio cognome, il risultato che vedremo nella console del browser sarà:



Figura 13.8 – Observable prodotto dalla digitazione nel campo nome

Questo mostra chiaramente la differenza tra `value` e `valueChanges`. Le modifiche apportate al campo infatti non sono intercettate da `value`, a meno che non lo faccia alla fine del riempimento di tutti i dati del modulo, ossia con la classica azione di `submit` del modulo che vedremo a breve.

Fino ad ora abbiamo lavorato sul corpo della classe del componente. Ma se volessimo visualizzare le modifiche apportate al campo direttamente nel template, per vedere in azione il meccanismo del “Data Binding” ottenuto con la direttiva `formControlName`, cosa dobbiamo usare?

Si può sfruttare , o lo stesso codice visto per recuperare lo snapshot, grazie alla proprietà `value`, oppure direttamente il nome del controllo presente nel modulo , a

patto di inserire un alias all'interno del corpo della classe grazie a un getter. Qui sotto vediamo applicate entrambe le tecniche al campo nome:

```
<p>Valore in Nome: {{ mioForm.get('nome').value }} </p>
<p>Valore in Nome: {{ nome }} </p> // solo con un getter
```

Ad ogni pressione del tasto , generiamo un evento, che attiva un nuovo ciclo di update della vista, che recupererà il nuovo valore inserito dall'utente e lo visualizzerà all'utente, ottenendo lo stesso risultato visto nei form realizzati con la tecnica “Template Driven” che sfruttavano la direttiva `ngModel` e la notazione “banana in a box”.

login/form-iscriviti.component.ts

```
//...
import {FormGroup, FormControl, FormBuilder} from '@angular/forms';
@Component({
  selector: 'ca-form-iscriviti',
  template:
    `
      <form [formGroup]= "mioForm">
        <label for="nome">Nome</label>
        <input type="text" placeholder="Il tuo nome"
               formControlName="nome">
        <label for="email">Email</label>
        <input type="email" placeholder="La tua email"
               formControlName="email">
        <button type="submit">Invia</button>
      </form>
      <p>Valore in Nome: {{ nome.value }} </p>
      <p>Valore in Email: {{ email.value }} </p>
    `
})
export class FormIscrivitiComponent implements OnInit {
  private mioForm: FormGroup;
  private nomeObs: string;

  constructor(private fb: FormBuilder) {
    this.mioForm = this.fb.group({
      nome: 'Davide',
      email: ''
    });
  }
  ngOnInit() {
    console.log("Nome inserito è " + this.mioForm.get('nome').value);
    this.mioForm.get('nome').valueChanges
      .subscribe(tasto => {
        console.log("Tasto premuto è: " + tasto);
        this.nomeObs = tasto;
      });
  }
}
```

(continua)

```
// definisco un getter per la proprietà nome
get nome():string {
    return this.mioForm.get('nome').value;
}

// definisco un getter per la proprietà email
get email():string {
    return this.mioForm.get('email').value;
}
}
```

13.13 Recuperare il valore dei campi dopo il submit

Quello che ci manca ora per chiudere il cerchio è vedere come recuperare, in una sola volta, tutti i dati dei vari campi del form, dopo che l'utente ha cliccato sul pulsante invio.

Si sfrutta anche in questo caso l'evento `ngSubmit` che ci permetterà di definire il gestore d'evento.

A quest'ultimo non è necessario passare alcun parametro, visto che l'istanza del form è stata creata direttamente nel corpo della classe.

```
//...
<form [formGroup]= "mioForm" (ngSubmit)="invio()">
    <label for="nome">Nome</label>
    <input type="text" placeholder="Il tuo nome"
           formControlName="nome">
    <label for="email">Email</label>
    <input type="email" placeholder="La tua email"
           formControlName="email">
    <button type="submit">Invia</button>
</form>
<p>Valore in Nome: {{ nome.value }} </p>
<p>Valore in Email: {{ email.value }} </p>
//...
```

All'interno del gestore d'evento `invio()` possiamo recuperare tutti i valori dei campi sotto forma di oggetto con chiave corrispondente al nome del `formControl` e valori pari al dato inserito dall'utente e che possiamo usare per valorizzare un tipo creato ad hoc , con ulteriori campi che aggiungeremo nei prossimi paragrafi.

```
model/utente.model.ts
```

```
export interface Utente {
  nome:string;
  email: string;
  //...
}
```

In questo modo lo possiamo agevolmente passare a future funzioni, senza preoccuparci di valorizzare i singoli campi .

```
login/form-iscriviti.component.ts
```

```
//...
export class FormIscrivitiComponent implements OnInit {
  private mioForm:FormGroup;
  utente: Utente;

  constructor(private fb: FormBuilder) {
    this.mioForm = this.fb.group({
      nome: 'Davide',
      email: ''
    });
  }

  invio() {
    console.log("Dati del form" + JSON.stringify(this.mioForm.value));
    this.utente = this.mioForm.value;
  }
}
```

Il messaggio che visualizzeremo nella console del browser sarà un oggetto JavaScript con le proprietà nome ed email :



Figura 13.9 – Oggetto prodotto dall’invio dei dati del form

Mentre se volessi visualizzare il tutto all’interno della vista del componente , dovremmo sfruttare il pipe json e scrivere:

```
<p>Valori inseriti nel Form {{ mioForm.value | json }} </p>
```

Prima di passare ad analizzare la gestione delle altre tipologia di campi di un form, vediamo come implementare le basi per effettuare i controlli di validità dei valori immessi all'interno di un campo.

Il passo successivo dopo l'invio è quello di pulire i campi del form, in modo che l'utente possa inserirne di nuovi nel caso non siano corretti in seguito a specifici controlli che vedremo nel prossimo paragrafo.

Per questo possiamo sfruttare il metodo reset della classe `FormGroup`.

```
login/form-iscriviti.component.ts
//...
invio() {
  console.log("Dati del form" + JSON.stringify(this.mioForm.value));
  this.utente = this.mioForm.value;
  this.mioForm.reset({
    nome : '',
    email : ''
  });
}
//...
```

13.14 Gestire i controlli dei campi di un form

All'atto della costruzione del form abbiamo usato la classe `FormBuilder` e il metodo `group` per creare le singole istanze dei vari campi.

Oltre a inizializzarli con un valore, è possibile sfruttare la classe `Validators`, disponibile sempre nella libreria `@angular/forms`, per inserire un primo controllo di validità dei valori immessi dall'utente.

Chiaramente potrei sfruttare direttamente i controlli predefiniti dell'HTML ma per una più semplice integrazione con le funzionalità offerte da Angular, conviene aggiungerli direttamente all'atto della costruzione delle istanze dei vari `FormControl`, potendoli poi eventualmente personalizzare.

In che modo? Passando delle funzioni di validazione, come secondo argomento, al `FormControl` che definisce il modello di ogni campo del form.

In questo modo, ad ogni nuovo valore inserito, viene richiamata la relativa funzione ed effettuato il controllo. Il tutto lavorando direttamente nel corpo della classe del componente e in modalità sincrona.

Ad esempio se volessimo rendere obbligatoria la compilazione di entrambi i campi, possiamo sfruttare la funzione `Validators.required`.

```
this.mioForm = this.fb.group({
  nome: ['Davide', Validators.required],
  email: ['', Validators.required]
});
```

Esistono altre funzioni di controllo predefinite che è possibile usare , come evidenziato qui sotto:

- Validator.required** → rende obbligatorio il campo
- Validator.requiredTrue** → rende obbligatorio il campo se i valori sono booleani
- Validator.minLength(N)** → imposto un minimo numero di caratteri obbligatori
- Validator.maxLength(N)** → imposto un massimo numero di caratteri obbligatori
- Validator.email** → controlla che il campo sia un'email valida

Non dobbiamo per forza limitarci ad una singola funzione per elemento del modello, ma possiamo aggiungerne in cascata con questa sintassi:

```
this.mioForm = this.fb.group({
  nome: ['Davide', [Validators.required, Validators.minLength(3)]],
  email: ['', [Validators.required, Validators.email]]
});
```

Per segnalare all'utente il motivo per cui non è valido il dato inserito, dobbiamo sfruttare la proprietà `invalid` e la direttiva `*ngIf` al fine di mostrare o meno la sezione con il messaggio d'errore.

Analogo discorso per il bottone d'invio, dove possiamo sfruttare l'attributo `disabled`.

```
//...
<form [formGroup]= "mioForm" (ngSubmit)="invio()">
  <label for="nome">Nome</label>
  <input type="text" placeholder="Il tuo nome"
         formControlName="nome">
  <label for="email">Email</label>
  <input type="email" placeholder="La tua email"
         formControlName="email">
  <button type="submit" [disabled]="mioForm.invalid">Invia</button>
</form>
<div *ngIf="mioForm.get('nome').invalid">
```

(continua)

```

        Campo Nome obbligatorio
    </div>
<div *ngIf="mioForm.get('email').invalid">
    Campo Email obbligatorio
</div>
//...

```

Questa però non è sufficiente o meglio non fornisce la migliore esperienza utente, in quanto non appena visualizziamo il form nel browser compare anche il messaggio “Campo Email Obbligatorio”.

The screenshot shows a simple form with two fields. The first field is labeled "Nome" and contains the value "Davide". The second field is labeled "Email" and has the placeholder text "La tua email". Below the fields is a green button labeled "INVIA". Above the "Email" field, the text "Campo Email obbligatorio" is displayed in red, indicating an invalid input. This message appears even though the "Nome" field is valid.

Figura 13.10 – Visualizzazione prematura del messaggio sul campo obbligatorio

Questo è dovuto al fatto che il campo nome è già stato valorizzato durante la creazione, mentre il campo email non presenta alcun valore. Dobbiamo sfruttare altre proprietà della classe `AbstractControl`, tra cui:

- **dirty** → true se l'utente ha modificato il valore nell'interfaccia utente
- **touched** → true se l'utente ha attivato un evento “blur” su di esso (es. ci ha cliccato sopra)
- **pristine** → true se l'utente non ha ancora modificato il valore nell'interfaccia utente.

Ognuna di queste proprietà attiva anche il corrispondente attributo di classe nel codice HTML del form visto in figura 13.6 (`ng-dirty`, `ng-touched`, `ng-pristine`). Ciò significa che possiamo creare delle regole CSS personalizzate per cambiare lo stile del form e l'effetto visivo delle segnalazioni da mostrare all'utente.

Una possibile modifica del form potrebbe basarsi sul fatto che il campo non è valido se è stato in qualche modo selezionato o se l'utente ha inserito dei dati. In questa ulteriore condizione ha senso mostrare il messaggio all'utente:

```
//...
<form [formGroup]= "mioForm" (ngSubmit)="invio()">
    <label for="nome">Nome</label>
    <input type="text" placeholder="Il tuo nome"
           formControlName="nome">
    <label for="email">Email</label>
    <input type="email" placeholder="La tua email"
           formControlName="email">
    <button type="submit" [disabled]="${mioForm.invalid}">Invia</button>
</form>
<div *ngIf="!nome.valid && (nome.dirty || nome.touched)">
    Campo Nome obbligatorio
</div>
<div *ngIf="!email.valid && (email.dirty || email.touched)">
    Campo Email obbligatorio
</div>
//...
```

Questi stessi messaggi, limitati solo al check dell'obbligatorietà di un campo, potranno essere usati anche nel caso di form creati con la tecnica “Template Driven” vista all'inizio del capitolo, potendo referenziare i singoli campi grazie alla variabile template inserita in ogni campo.

Ora che tramite il modello creato, sappiamo sia inizializzare i campi di un form, sia recuperare i singoli valori, vediamo come l'operazione di aggiornamento di un campo, ad esempio in seguito al recupero dei dati memorizzati in precedenza all'interno di un database remoto.

Esercitazione facoltativa: potresti provare a riprogettare la sezione di login dell'applicazione MetroChat, creata nell'esercitazione precedente, sfruttando le tecniche reattive.

13.15 Settare un dato all'interno di un campo di <input>

L'aggiornamento di un campo di input in seguito all'interrogazione di un'API che fornisce il dato precedentemente inserito dall'utente e memorizzato su un database, può essere fatta con il metodo `setValue()` che chiaramente agisce sul FormControl.

Ipotizzando di voler aggiornare il campo Nome del modulo web tramite un metodo `updateForm()`, richiamato all'interno del metodo `ngOnInit()`, potremmo scrivere:

login/form-iscriviti.component.ts

```
//...
export class FormIscrivitiComponent implements OnInit {

private mioForm: FormGroup;
private nomeObs: string;
private utente: User;

constructor(private fb: FormBuilder) {
    this.mioForm = this.fb.group({
        nome: ['Davide', [Validators.required, Validators.minLength(3)]],
        email: ['', [Validators.required, Validators.email]]
    });
}

// definisco un getter per la proprietà nome
get nome():string {
    return this.mioForm.get('nome').value;
}

// definisco un getter per la proprietà email
get email():string {
    return this.mioForm.get('email').value;
}

ngOnInit() {
    this.updateForm();
}

updateForm() {
    this.nome.setValue('Luca');
}

invio() {
    console.log("Dati form " + JSON.stringify(this.mioForm.value));
    this.utente = this.mioForm.value;
    console.log("Nome: " + this.utente.nome);
    console.log("Email: " + this.utente.email);
}
}
```

Fai attenzione che ogni chiamata al metodo `setValue()` produce un ciclo di update della vista.

In alternativa potresti aggiornare direttamente tutti i campi del form accedendo direttamente all’istanza di `formGroup` e passando l’oggetto che rappresenta tutti i campi del modulo.

```
// ...
updateForm() {
  this.mioForm.setValue({
    nome: 'Luca',
    email: 'lucarossi@miaemail.com'
  });
}
//...
```

Nella pratica, quello che ti capiterà più frequentemente, sarà l’aggiornamento parziale solo di alcuni campi del modulo.

In questo caso possiamo usar e un altro metodo `patchValue()`, sempre applicato all’istanza `FormGroup`, a cui passeremo l’insieme delle proprietà e i rispettivi valori aggiornati, sotto forma di oggetto JavaScript parziale .

```
//...
updateForm() {
  this.mioForm.patchValue({
    nome: 'Luca',
    email: 'lucarossi@miaemail.com'
  });
}
//...
```

L’effetto risultante è che ora nei campi del modulo web vedremo entrambi i campi con dei nuovi valori. E’ chiaro che avendo solo due campi avremmo potuto applicare direttamente `setValue()`.

13.16 Gestire i dati di un campo di <select>

Ipotizzando di voler inserire nel modulo lo stesso campo provincia visto nella prima parte del capitolo, il primo passo sarà “collegare” il campo del form, con il controllo che creeremo nel corpo della classe.

Limitandoci solo a questo nuovo campo per focalizzare l’obiettivo, potremmo scrivere:

```
<form [formGroup]="mioForm" (ngSubmit)="invio()">
  <label for="pv">Provincia</label>
  <select name="pv" formControlName="pv">
    <option *ngFor="let pv of prov" value="{{pv.valore}}">
      {{ pv.nome }}
    </option>
  </select>
  <button type="submit" [disabled]="mioForm.invalid">Invia</button>
</form>
```

Nota la presenza dell'attributo value che ci serve per inserire un valore all'interno dei vari tag option. Il modello che creeremo nella classe del componente, dovrà avere lo stesso nome, ossia pv:

```
//...
constructor(private fb: FormBuilder) {
  this.mioForm = this.fb.group({
    nome: ['Davide', [Validators.required, Validators.minLength(3)]],
    email: ['', [Validators.required, Validators.email]],
    pv: ['', Validators.required]
  });
}
//...
```

Qualsiasi selezione effettueremo nel relativo campo del modulo HTML della pagina, potrà essere intercettata con le stesse tecniche viste per il campo di input.

```
//...
invio() {
  console.log("Dati form " + JSON.stringify(this.mioForm.value));
  console.log("Solo dato Select: " + this.mioForm.get('pv'));
  this.utente = this.mioForm.value;
  this.mioForm.reset({
    nome : '',
    email : ''
  });
}
//...
```

13.17 Settare un valore all'interno di un campo <select>

L'operazione di settaggio di un campo di `select`, segue le stesse regole dei campi di `input`, ossia dovremo o agire direttamente sul singolo controllo tramite `setValue()` oppure, a più ampio spettro , con `patchValue()` indicando il nome del controllo e il valore.

Ad esempio se nei dati recuperati dall'API avessimo memorizzato il valore associato alla provincia di Bologna, quindi "BO", l'aggiornamento potrebbe essere apportato con queste righe:

```
//...
updateForm() {
  this.mioForm.patchValue({
    nome: 'Luca',
    email: 'lucarossi@miaemail.com',
    pv: 'BO'
  });
}
//...
```

13.18 Gestire i dati di un campo <checkbox>

Continuando con le analoghe modifiche fatte nella prima parte del capitolo, nel caso volessimo gestire un campo `checkbox`, il primo passo come sempre è collegare i campi del form, con il modello che creeremo nel corpo della classe, sfruttando la direttiva `formControlName`.

```
<form [formGroup]="mioForm" (ngSubmit)="invio()">
  <label for="catA">Categoria A</label>
  <input type="checkbox" formControlName="catA">
  <label for="catB">Categoria B</label>
  <input type="checkbox" formControlName="catB">
  <button type="submit" [disabled]="mioForm.invalid">Invia</button>
</form>
```

Il modello che creeremo nella classe del componente, dovrà avere lo stesso nome, ossia rispettivamente `catA` e `catB`.

Come valori, potremmo continuare a usare una stringa vuota, ma non sarebbe corretto nel senso che i valori ammessi sappiamo sono o `true` o `false`, che corrispondono rispettivamente al campo selezionato o non selezionato.

Nel caso volessimo rendere obbligatorio solo il primo campo, non potremmo più usare la funzione `required`, ma `requiredTrue`:

```
//...
this.mioForm = this.fb.group({
  nome: ['Davide', [Validators.required, Validators.minLength(3)]],
  email: ['', [Validators.required, Validators.email]],
  pv: ['', Validators.required],
  catA: [false, Validators.requiredTrue],
  catB: [false],
});
//...
```

Indipendentemente dallo stato, il recupero del valore `true` o `false` all'interno della classe, avviene con la stessa notazione vista per gli altri campi:

```
//...
invio() {
  console.log("Dati form " + JSON.stringify(this.mioForm.value));
  console.log("Cat. A :" + this.mioForm.get('catA').value);
  console.log("Cat. B :" + this.mioForm.get('catB').value);
}
//...
```

Per impostare il campo allo stato selezionato, dopo aver recuperato il valore da un ipotetico database remoto, dovremo assegnare il valore `true` al relativo controllo. Possiamo sfruttare `setValue()` o più in generale `patchValue()`:

```
//...
updateForm() {
  this.mioForm.patchValue({
    catA : true,
    email: 'lucarossi@miaemail.com',
  });
}
//...
```

L'effetto che ottengo è che il campo categoria A è visualizzato nella stato selezionato.



Figura 13.11 – Campo checkbox impostato nello stato selezionato

13.19 Gestire i dati di un campo radio

Terminiamo il capitolo parlando del campo radio. Ipotizzando di usare sempre le due opzioni relative al sesso o Maschio o Femmina, la modifica da apportare al modulo web, saranno analoghe a quelle viste per il campo checkbox. Nota come, analogamente al campo select, sia stato inserito l'attributo value per contrassegnare il valore da recuperare.

```
<form [formGroup]="mioForm" (ngSubmit)="invio()">
  <label for="sesso">Sesso</label>
  <span>Maschio</span>
  <input type="radio" value="M" formControlName="sesso">
  <span>Femmina</span>
  <input type="radio" value="F" formControlName="sesso">
  <button type="submit" [disabled]="mioForm.invalid">Invia</button>
</form>
```

Nel corpo della classe inseriamo le righe per inizializzare il valore con una stringa vuota e rendiamo il campo obbligatorio:

```
//...
this.mioForm = this.fb.group({
  nome: ['Davide', [Validators.required, Validators.minLength(3)]],
  email: ['', [Validators.required, Validators.email]],
  pv: ['', Validators.required],
  catA: [false, Validators.requiredTrue],
  catB: [false],
  sesso: ['', Validators.required]
});
```

Il recupero del valore “M” o “F”, anche in questo caso avviene con la stessa notazione vista per gli altri campi:

```
//...
invio() {
  console.log("Dati form " + JSON.stringify(this.mioForm.value));
  console.log("Sesso: " + this.mioForm.get("sesso").value);
}
//...
```

Per impostare invece il campo allo stato selezionato, dovremo assegnare il corrispondente valore definito nell'attributo value presente nel template. Possiamo sfruttare setValue() o più in generale patchValue():

```
//...
updateForm() {
  this.mioForm.patchValue({
    sesso: 'M',
    email: 'lucarossi@miaemail.com',
  });
}
//...
```

L'effetto che ottengo è che il campo sesso sarà visualizzato nella stato selezionato .



Figura 13.12 – Campo radio impostato nello stato selezionato

Capitolo 14

Le basi della libreria RxJS

14.1 Dalla programmazione asincrona agli Observable

Arrivati a questo punto del libro, gran parte dell'applicazione è realizzata e funzionante. Gran parte dei concetti Angular che userai nell'80% delle applicazioni ora ti è noto. Mancano tuttavia all'appello una serie di ulteriori concetti e di approfondimenti che ho volutamente lasciato verso la fine per non appesantire troppo la trattazione.

Nozioni estremamente importanti che ti aiuteranno a capire molti dei concetti già visti in precedenza, ma senza quell'approfondimento necessario per riuscire a compiere un ulteriore passo in avanti nello sviluppo di applicazioni sempre più complesse.

Alla fine del capitolo applicheremo gran parte dei concetti direttamente all'applicazione sviluppata fino ad ora, ma dovremo ogni tanto sganciarci da quel codice per rendere gli esempi più specifici e focalizzati al concetto da apprendere.

Tutte le azioni che facciamo su un'interfaccia utente (clic, tocco, inserimento), tutte le chiamate ad API remote che un'applicazione esegue, tutti gli eventuali errori da segnalare all'utente, sono tipiche sequenze di operazioni che avvengono in modo asincrono. Nessuno è in grado di stabilire a priori l'ordine di esecuzione.

Ad esempio quando l'utente apre per la prima volta l'applicazione MetroChat realizzata, in automatico attiva l'invio di una richiesta all'API responsabile di fornire i dati con la lista dei treni. Dobbiamo tuttavia attendere qualche istante prima di ricevere la risposta.

In questo arco di tempo, tra la richiesta e la risposta, grazie alla programmazione asincrona possa dire al programma di eseguire altre cose, senza necessariamente attendere la fine della precedente istruzione.

Analogo discorso se abbiamo progettato un'interfaccia utente con diversi bottoni ognuno specializzato per una particolare azione. Il clic sul primo potrebbe attivare l'invio di una richiesta di dati ad un'API, il click sul secondo l'apertura di un finestra di dialogo, e così via. Molto spesso poi il risultato prodotto da un'azione potrebbe innescare un'altra azione.

Il programmatore si trova a dover gestire un flusso di dati ed eventi, di cui non sa l'esatta sequenza d'arrivo. In alcuni casi l'ordine d'esecuzione è fondamentale, basti pensare alle fasi di autenticazione di un utente in un sito, in cui prima è necessario recuperare l'id dell'utente, poi interrogare una tabella per verificarne i requisiti di accesso e, se rispettati, dare l'accesso.

Analogo discorso se volessi inviare una richiesta ad un'API, solo dopo che si è eseguita una sequenza di azioni in successione (es. salvataggio dei dati immessi in un form, dopo che si sono modificati almeno 3 caratteri di un campo)

Nel caso ognuna di queste fase restituisse un a Promise, una possibile implementazione potrebbe essere di questo tipo:

```
this.loginService.getCurrentUserInfo()
  .then(info => {
    this.userService.getUserRights(info.userId)
      .then(rights => this.userRights = rights);
  });
}
```

Programmi con tante nidificazioni , oltre ad essere poco leggibili, spesso conducono ad errori di difficile individuazione.

Diventa allora fondamentale avere un insieme di concetti e nuovi operatori che agevolino il compito del programmatore nel progettare applicazioni asincrone caratterizzate dall'osservazione di flussi di dati o eventi emessi in modo asincrono.

La libreria RxJS (Reactive JavaScript) nasce quindi per colmare il vuoto lasciato della libreria nativa JavaScript nel gestire in modo efficiente , proprio sequenze di eventi o di dati asincroni. Non è da escludere che nel futuro diventi parte delle API standard di JavaScript visto che già in ES7 si è iniziato a parlare di questi concetti e ES10 sono apparsi i primi operatori .

Usare la libreria RxJS significa passare da una programmazione imperativa in cui il valore di una data variabile viene modificato con una classica istruzione di assegnazione in un preciso momento del flusso del programma, a una programmazione Reattiva in cui l'assegnazione del valore non è fatta esplicitamente

dal programmatore, ma avviene in conseguenza al cambio di una dato verificatosi in un altro punto del programma.

Tale libreria è stata sviluppata anche per altri linguaggi come Java con RxJava, PHP con RxPHP, Scala con RxScala ecc.

I concetti chiave che dovrai imparare sono essenzialmente cinque, e per ognuno è necessario conoscere una terminologia del tutto nuova. Per ora non è importante capirli a fondo, ma solo intuirne alcune peculiarità.

- **Observable:** rappresentazione di una serie di valori o eventi futuri emessi nel tempo.
- **Observer:** collezione di funzioni di richiamata (callbacks) che sanno come gestire i valori inviati da un Observable .
- **Subscription:** rappresenta l'azione di esecuzione di un Observable, ed è utile per terminare l'esecuzione del flusso di dati.
- **Operatori:** sono classiche funzioni che permettono di manipolare i dati emessi dall'Observable.
- **Subject:** è simile ad un EventEmitter, e permette di trasmettere di creare Hot Observable, ossia emettere un valore o evento a più Observable contemporaneamente.
- ***Schedulers:** permettono di creare “Multithread” con “dispatchers” centralizzati usati per calendarizzare operazioni da effettuare nel futuro senza preoccuparci di programmare ogni chiamata (es. operazioni che richiedono molte risorse, sono spostate su un thread dedicato, per poi passare al thread principale una volta terminate)

Ognuno di questi termini ti diventerà familiare non appena avrai terminato il capitolo quindi non soffermiamoci più di tanto sul significato di ognuno. In particolare l'ultimo non lo userai spesso nelle comuni applicazioni, se non durante alcune fasi di test del codice . Facciamo per ora solo un paio di considerazioni sul primo termine: Observable.

Lo potremmo tradurre dall'inglese in “Osservabile”. In effetti la prima volta che l'ho incontrato, devo essere sincero, ho fatto un po' di fatica a intuire il reale significato. Conoscevo il termine “Promise” e pensavo fosse sufficiente per descrivere dati asincroni. In realtà mi sbagliavo.

Il concetto di Promise permette di descrivere un singolo dato futuro asincrono. Quando invece dobbiamo gestire più dati o eventi asincroni, il concetto di Promise non è più sufficiente e deve essere, in un certo senso, esteso.

La seconda considerazione è che se memorizzi subito che un Observable è a tutti gli effetti una “funzione”, ti risparmierai molta della fatica fatta da me nel cercare di capire il reale significato e soprattutto l’uso che si fa nelle applicazioni Angular.

Ho scritto funzione tra virgolette, perché gli Observable hanno delle caratteristiche non presenti nelle normali funzioni.

Le funzioni che tutti noi conosciamo, infatti, una volta definite, possono restituire un valore o più valori contemporaneamente. Inoltre se vogliamo far emettere nuovi valori, dobbiamo nuovamente richiamarle.

Gli Observable invece sono paragonabili a funzioni in grado di emettere più valori o eventi nell’arco di un certo periodo di tempo, in modo asincrono, senza che per ogni emissione debbano essere esplicitamente richiamate nel codice, come avviene nella programmazione imperativa.

E’ chiaro che, come tutte le funzioni, ci deve essere una prima chiamata, ma poi continuano ad emettere i dati sulla base della logica sviluppata internamente ad essi.

Definire un Observable è paragonabile all’acquisto di un lanciatore automatico, quello che usano i tennisti per allenarsi. La sequenza di palline da tennis che viene lanciata nel tempo, è il risultato dell’accensione della macchina che genera un Observable, ossia un flusso di palline lanciate con cadenza variabile nel tempo.

Tutte le applicazioni che creerai in Angular avranno sicuramente dei flussi di dati o eventi di questo tipo, quindi potranno essere tutti rappresentati con Observable.

Dal flusso di dati inseriti dall’utente in un campo di input, a quelli generati da una chiamata ad un’API a quelli generati dallo spostamento di un oggetto nello schermo e via dicendo.

Pertanto ogni volta che tra le righe del nostro programma incontreremo un Observable, è come se stessimo leggendo un’istruzione simile a questa:

```
let dati = function() { ... } // in JavaScript
```

dove dati è la definizione del nostro ipotetico Observable.

Quando creiamo un Observable, definiamo a tutti gli effetti una particolare funzione che ha la peculiarità di sfornare dei dati o eventi nel tempo.

Nel resto dei prossimi capitoli useremo più spesso il termine dato, ma le stesse considerazioni varranno anche per gli eventi.

Visto che sono nati per rappresentare flussi di dati o eventi nel tempo, un Observable lo puoi pensare come alla definizione di quali dati o eventi saranno emessi dalla funzione non appena questa sarà eseguita.

Una differenza sostanziale rispetto alle classiche funzioni JavaScript è che per gestire le informazioni da inviare in uscita, l'Observable usa una serie di funzioni (callback), adibite a inviare non solo i dati, ma a segnalare errori, o a segnalare il termine del flusso di dati.

La domanda che sorge spontanea è: come facciamo a recuperare i dati emessi nel tempo?

Come dicevamo in precedenza, non appena creiamo un Observable, questo non emetterà alcuna informazione perché è necessario che in qualche punto del programma qualcuno lo “invochi”.

Se per invocare una funzione basta inserire un richiamo al proprio nome, nel caso degli Observable è necessario effettuare l'operazione d'“iscrizione” o, come viene chiamata in gergo, l'operazione di “subscribe”.

Fino a che non si effettua questa operazione, l'Observable non invierà alcuna informazione. La puoi pensare come all'abbonamento a un servizio per la visualizzazione dei video online (es. la nostra WebSU). Fino a che non ti iscrivi, non potrai visualizzare i video che abbiamo preparato per te.

Pertanto tornando all'analogia con la definizione di una funzione, se per poterla eseguire dobbiamo richiamarla:

```
let dati = function () {...}
let result = dati();
```

nel caso degli Observable, dovremo effettuare l'operazione di subscribe, richiamando il relativo metodo:

```
let dati$ = .... // definisco un Observable
let subscriber = dati$.subscribe();
```

Il metodo `subscribe()` fa parte della libreria RxJS, ed è uno dei tanti metodi che prendono il nome di operatori RxJS , nel senso che ci permettono di operare sui dati emessi dall'Observable. In realtà non è propriamente un operatore, come quelli che vedremo in seguito, ma giusto per rendere l'idea.

La libreria RxJS ha più di cento operatori , ognuno con una precisa specificazione: da quelli che creano Observable, a quelli che trasformano i dati, a quelli che filtrano i dati ecc.

Per tua fortuna nelle applicazioni Angular, che non siano dei giochi, raramente ne userai più di una decina, quindi non prendere paura: non dovrai impararli tutti e cento a memoria.

14.2 Peculiarità di un Observable

Ora che abbiamo capito a grandi linee alcuni dei primi concetti chiave della libreria RxJS, vediamo di entrare nel vivo dei dati. Gli Observable come dicevamo all'inizio, nascono per cercare di rappresentare e gestire al meglio flussi di dati o eventi asincroni che si generano in ogni applicazione.

A volte l'uso di un grafico permette di comprendere meglio i concetti. Si sfrutta per questo un diagramma, chiamato diagramma di Marble, che ha queste sembianze.



Figura 14.1 – Diagramma di Marble per rappresentare i valori emessi da un Observable

La linea retta orizzontale indica il passare del tempo. Per indicare la fine di emissione con successo del flusso di dati dell'Observable, si usa una linea etta perpendicolare alla retta, mentre si usano i cerchi per rappresentare i singoli dati emessi nel tempo dall'Observable.

Tutta la documentazione ufficiale che troverai nel sito RxJS fa uso di questo diagramma combinato con gli operatori, che sono rappresentati con dei rettangoli sotto il flusso di dati.

Per indicare l'interruzione non prevista di un flusso di dati Observable, si usa inserire una lettera X sopra alla linea.

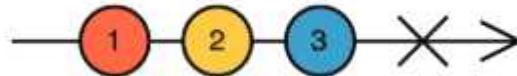


Figura 14.2 – Diagramma di Marble con indicazione di un flusso interrotto

L'applicazione di un operatore, genera sempre un nuovo flusso di dati, quindi un altro Observable, che avrà delle caratteristiche diverse dall'originale, a seconda del funzionamento dell'operatore applicato.

L'Observable sorgente quindi non viene mai modificato dall'applicazione di un operatore.

Capito come può essere rappresentato a livello grafico vediamo come si può definire a livello di codice. Dovremo rispondere alla domanda: come si definisce in TypeScript una variabile che dovrà contenere un Observable?

Per convenzione si fa seguire il nome dal simbolo di \$ (non ne conosco le motivazioni esatte, ma a me ricorda il concetto di flusso di denaro). Poi si fa seguire la parola Observable e si indica il tipo di dati di cui sarà costituita.

```
listamessaggi$ : Observable<TIPO>
```

NB: Questa variabile non è la classica variabile a cui siamo abituati nel senso che è solo la definizione di un Observable. Non potrò assegnargli direttamente una valore ma dovrò prima effettuare l'operazione di subscribe (abbonamento) un po' come avviene quando vuoi guardare dei film. Prima devi iscriverti al servizio.

Quando ad esempio nel capitolo 12 abbiamo progettato il metodo `getListaMetroObservable()`, abbiamo indicato come dato restituito, proprio un Observable con tipo array di oggetti Metro:

```
getListaMetroObservable(): Observable<Metro[]> {...}
```

Nei diversi componenti che sfruttano questo service, potremmo quindi definire una variabile Observable, valorizzata con la chiamata al metodo:

```
listametro$: Observable<Metro[]>;
listametro$ = this.treniservice.getListaMetroObservable();
listametro$.subscribe(
  risp => this.listametro = risp,
  error => this.errormsg = error
);
```

Rispetto alla sintassi usata nel Cap. 12, il vantaggio è che ora possiamo liberare le risorse allocate dall'app, non appena il componente viene distrutto, come avremo modo di discutere nei prossimi capitoli. Chiaramente questo è un modo, ma avremmo potuto ottenere lo stesso effetto sfruttando il pipe `async`.

Al metodo `subscribe()` abbiamo passato due callback, uno che gestisce i dati in arrivo e l'altro l'errore. Avrei comunque potuto inserire anche il terzo callback per gestire tutto quello da eseguire dopo o l'arrivo dei dati o l'arrivo di un errore.

Se infatti osserviamo la definizione base del metodo `subscribe()` della classe Observable ricavata dalla libreria RxJS, notiamo subito la struttura dei parametri richiesti.

```
class Observable<T> implements Subscribable {
  static create: Function
  static if: typeof iif
  static throw: typeof throwError
  subscribe(observerOrNext?: NextObserver<T> | ErrorObserver<T> | CompletionObserver<T> | ((value: T) => void),
            error?: (error: any) => void,
            complete?: () => void): Subscription...
```

Come vedi ha diverse possibili firme, ma in generale viene usata quella che prevede il passaggio di un oggetto Observer, dal nome della relativa interfaccia.

Un Observable senza un Observer è come il player di un video senza il pulsante di play. E' necessario quindi fornire sempre almeno il callback che gestisce i dati emessi dall'Observable. Generalmente in Angular si preferisce passare tre funzioni come argomento, ognuna adibita ai tre compiti detti: ricezione, errore, completamento.

```
subscribe((value: T) => void),
    error?: (error: any) => void,
    complete?: () => void):Subscription...
```

In alternativa si potrebbe creare un oggetto che implementa l’interfaccia Observer e quindi i relativi metodi `next()`, `error()` e `complete()`.

Queste considerazioni chiaramente sono valide anche nel caso di eventi. IN JavaScript ad esempio se vogliamo iniziare a monitorare i click sul documento, dobbiamo aggiungere un gestore d’evento e una funzione che esegua delle operazioni sugli eventi ricevuti .

```
eventi = document.addEventListener('click', (evt) => {
    console.log(evt)
});
```

Nel caso degli Observable, si dice che bisogna iscriversi all’Observable per iniziare a recuperare le informazioni sugli eventi. A analogo discorso per rimuovere il gestore d’evento. Nel caso degli Observable di parlerà di cancellazione o “unsubscribe” dell’Observable.

```
streamclick$ = fromEvent(document,'click'); // Observable
streamclick$.subscribe();
```

Nella prima riga, abbiamo definito l’Observable `eventi$`, ottenuto sfruttando l’operatore `fromEvent` della libreria RxJS.

Il primo argomento che richiede è dove si verifica l’evento, mentre il secondo è il tipo di evento che vogliamo monitorare.

Nel nostro caso sarà l’evento nativo ‘click’ o ‘keyup’ etc che si verificherà sull’intero documento.

Poi ho effettuato il `subscribe` , quindi ho fatto partire l’ascolto dei clic. Al fine di ricevere e gestire le informazioni sugli eventi emessi, devo passare al metodo almeno un callback:

```

streamclick$: Observable<Event> = fromEvent(document, 'click');
streamclick$.subscribe(
    value => console.log("Ho cliccato"), // obbligatorio
    err => console.log(),
    () => console.log()
);

```

Così facendo, la variabile value, conterrà di volta in volta le informazioni sui singoli eventi emessi nel tempo, che potrò elaborare con il relativo callback.

Chiaramente nessuno ci impedisce di aggiungere anche gli altri due callback, per gestire gli errori e per gestire il completamente del flusso dei dati.

La terza funzione è chiamata sempre senza argomenti e viene eseguita quando il flusso di dati/eventi emessi dall'Observable è terminato.

Ad esempio tutti i metodi della classe HttpClient che abbiamo visto nei capitoli precedenti, restituiscono un Observable che termina non appena è stato ricevuto.

L'eliminazione di messaggi in popup visualizzati durante la fase di attesa di risposta di un'API o dopo il verificarsi di un errore , tipicamente viene fatta proprio all'interno del terzo callback.

Per confermare che senza l'azione di `subscribe()` non è possibile ricevere alcun evento emesso dall'Observable, potremmo inserirlo all'interno di una condizione.

Ad esempio se volessimo attivare il monitoraggio del flusso di eventi clic, solo dopo che sono passati tre secondi, possiamo scrivere:

```

count = 0;
streamclick$: Observable<Event> = fromEvent(document, 'click');
setInterval(() => {
    this.count++;
    console.log(this.count);
    if (this.count === 3) {
        streamclick$.subscribe(
            value => console.log("Ho cliccato"), // obbligatorio
            err => console.log(),
            () => console.log()
        );
    }
}, 2000);

```

Solo dopo che la variabile count, ha raggiunto il valore 3, potrò intercettare i click fatti dall'utente sulla pagina. Tutti i clic fatti nei secondi precedenti, non saranno intercettati. Un po' come guardare un video online su Youtube senza avere il computer connesso alla rete. Puoi fare mille volte click su Play ma il video non partirà.

Nella figura qui sotto, si vede come i clic siano intercettati solo dopo che la variabile count ha assunto valore pari a 3.

```

Angular is running in the development mode. Call enableProdMode()
1 - continua a cliccare...
2 - continua a cliccare...
3 - continua a cliccare...
  click { target: html <div> , buttons: 0, clientX: 416, clientY: 134 }
  click { target: html <div> , buttons: 0, clientX: 416, clientY: 134 }
  click { target: html <div> , buttons: 0, clientX: 416, clientY: 134 }
4 - continua a cliccare...

```

Figura 14.3 – Iscrizione condizionata all’Observable che monitorizza i clic sul documento

Questa tipologia di Observable viene anche detta in gergo “Unicast” o “Cold”. Per adesso sorvoliamo sul significato preciso ma tieni a mente il paragone con la visualizzazione di un video gratuito su YouTube.

Per terminare la sezione, così come ci si iscrive all’Observable per ricevere il dato o l’evento da osservare, analogamente ci si può cancellare e interrompere quindi l’analisi del flusso di dati per liberare risorse usate dall’applicazione ed evitare che l’app si divori tutta la memoria a disposizione.

Infatti ogni volta che usiamo il metodo `subscribe()` generiamo un oggetto di tipo `Subscriber` che ha associato il metodo `unsubscribe()`. Sarà sufficiente quindi creare un riferimento a quest’ultimo durante l’iscrizione così come fatto in precedenza con l’Observable `listametro$`.

```

count = 0;
streamclick$: Observable<Event> = fromEvent(document, 'click');
clicksubscriber: Subscriber;
setInterval(() => {
  this.count++;>
  (continua)

```

```

        console.log(this.count);
        if (this.count === 3) {
            clicksubscriber = streamclick$.subscribe(
                value => console.log("Ho cliccato"),
                err => console.log(),
                () => console.log()
            );
        }
    }, 2000);
}

```

Nelle applicazioni Angular, se l'azione di sottoscrizione all'Observable è eseguita all'interno di un componente, e non ci sono altre condizioni per interrompere l'iscrizione, conviene usare l'aggancio al ciclo di vita `ngOnDestroy` che chiaramente dovrà essere implementato (`OnDestroy`) nella definizione della classe.

La variabile `this.objSub` potrà richiamare in ogni istante il metodo `subscribe()` per interrompere l'osservazione del flusso di dati. In alcuni casi questa operazione viene fatta in automatico da Angular, ma è preferibile farla manualmente per liberare tutte le risorse associate all'iscrizione.

```

ngOnDestroy() {
    this.clicksubscriber.unsubscribe();
}

```

In sostanza è come se stessimo dicendo che non ci interessa più alcun dato emesso dall'Observable.

Arrivati a questo punto potrebbero sorgerti mille domande.

Come si crea un Observable in Angular? Quali sono le tipiche classi in Angular che generano degli Observable? Come posso convertire un Promise in un Observable per sfruttare gli operatori della libreria RxJS?

Per ora rispondiamo alla prima domanda, perché le altre saranno oggetto della prossima sezione.

Nei precedenti capitoli abbiamo già usato più volte gli Observable, quando abbiamo introdotto le librerie `HTTPClient` e `Route`. La prima è quella che ci permette di interrogare un server remoto e ricevere uno o più valori nel tempo

sotto forma di flusso di dati (vedi Capitolo 10). Tutti i metodi di entrambe le classi, forniscono degli Observable, quindi potremmo fermarci qui e chiudere l'argomento avendo capito a grandi linee le caratteristiche più importanti.

In realtà, siamo solo all'inizio dell'esplorazione delle potenzialità della libreria quindi non perdiamoci d'animo e proseguiamo con le tecniche per creare degli Observable partendo dai più svariati dati.

14.3 Tecniche per creare Observable

Una delle domande che spesso mi sento rivolgere durante i corsi è perché è necessario sapere creare un Observable, se molte delle librerie Angular che si utilizzano nelle applicazioni, forniscono già dati in forma di Observable.

La risposta che spesso fornisco è: perché Angular ha le sue radici sulla programmazione in stile reattivo, e prima o poi ti capiterà di aggiornare l'interfaccia utente sulla base di un dato che si è modificato nel tempo.

Il metodo migliore per gestire un dato o un evento che cambia nel tempo è sfruttare gli Observable. Dovrai quindi conoscere le tecniche non solo per trasformare dati tipicamente statici come numeri, stringhe, array, ma anche per trasformare dati come Promise o altri dati restituiti in forma asincrona.

Fortunatamente la libreria RxJS mette a disposizione diversi metodi predefiniti (operatori) per generare degli Observable. Vediamone i principali di cui potresti aver bisogno.

La caratteristica peculiare è che sono tutti metodi applicabili quando conosciamo a priori i dati da trasformare, sia che siano eventi, numeri, stringhe, array. Al fine di poterli usare all'interno di un'applicazione Angular, è necessario importarli dalla libreria RxJS con una riga del tipo:

```
import { nomeoperatore } from 'rxjs';
```

Convertire una serie di eventi in un Observable

Per trasformare un flusso di eventi legati alle azioni del mouse, del tocco, o di input dell'utente, e in generale di un qualsiasi evento proveniente dal DOM, possiamo usare l'operatore RxJS `fromEvent()`.

```
import { fromEvent } from 'rxjs';
clicksEvt$: Observable<Event>;
clicksEvt = fromEvent(document, 'click');
clicksEvt.subscribe(x => console.log(x));
```

Convertire numeri, stringhe, oggetti in un Observable

Altro metodo spesso usato per trasformare una sequenza di numeri o un array in un Observable, è l'operatore `of()`. Ad esempio se volessi emettere una sequenza di stringhe con i valori che rappresentano i dati di accensione o spegnimento di un'apparecchiatura, potrei scrivere.

```
import { of } from 'rxjs';
seq$: Observable<string>;
seq$ = of('on', 'off');
seq$.subscribe(val => console.log("Stato: " + val));
```

Terminata l'emissione dell'ultimo dato, l'Observable termina. E' chiaro che quello che vedremo sarà una lista di stringhe emesse in modo sincrono quindi un blocco unico di dati, non cadenzato a intervalli temporali specifici o variabili.



Figura 14.4 – Visualizzazione dati emessi dall'Observable

Convertire array o collection in un Observable

Analoghe considerazioni varranno nel caso volessi trasformare un array o una collection. In questo caso si usa l'operatore `from`, che come `of()`, genera l'Observable e poi termina.

```
import { from } from 'rxjs';
seq$: Observable<string>;
seq$ = from(['on', 'on', 'off']); // trasformo l'array      (continua)
```

```
seq$.subscribe(val => console.log("Stato: " + val));
```

Convertire una Promise in un Observable

In molte situazioni avrai a che fare con funzioni che restituiscono delle Promise. Al fine di gestirle come se fossero degli Observable, puoi convertirle usando sempre l'operatore RxJS `from()`.

Potrebbe essere una chiamata `fetch()` ad un'API su un server remoto. (PS: Nelle vecchie versioni di RxJS si usava `fromEvent()`)

```
import { from } from 'rxjs';
seq$ = from(fetch('http://miosito.com/api/v1/'))
seq$.subscribe(val => console.log("Dati: " + val));
```

Convertire un Observable in un Promise

A volte può essere necessario convertire un Observable in un Promise. Capiterà raramente, ma in alcuni framework come Ionic, utilizzati per creare applicazioni ibride su dispositivi mobili, molti plugin restituiscono dati asincroni sotto forma di Promise, quindi a volte è più veloce trasformare la parte di codice che usa gli Observable in Promise. Per farlo possiamo sfruttare l'operatore `toPromise()`.

```
import { toPromise } from 'rxjs';
...
this.http.get('http://miosito.com/api/v1/')
    .toPromise()
    .then(dati => console.log("Dati: " + val));
```

Il metodo `get` della classe `HttpClient` restituisce un Observable, che trasformiamo in Promise potendo così utilizzare i classici operatori di concatenazione. Nota come in questo caso non sia necessario eseguire l'operazione di `subscribe`.

Creare un Observable senza operatori

Per avere un maggior controllo sull'Observable da creare, ci viene in aiuto la definizione della classe Observable, e in particolare il metodo `create()`. Tale

metodo richiede come parametro una funzione che sarà richiamata quando un Observer si iscriverà all'Observable.

L'Observer lo puoi pensare come alla classica formula che inserisci all'interno di una cella di un foglio Excel. Ogni volta che i dati delle celle a cui fa riferimento si modificano o emettono un nuovo valore, automaticamente è in grado di osservare il cambiamento e comunicarlo all'utente.

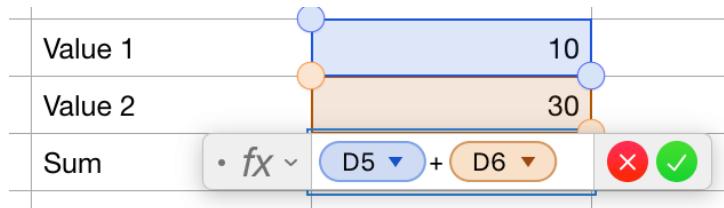


Figura 14.5 – Analogia tra Observer e formula di un foglio Excel

A sua volta il metodo `create()` riceverà come parametro l'istanza di un Observer, che ha agganciato i metodi `next()`, `error()` e `complete()`.

```
numeriRandom$ = Observable.create(observer => {
  observer.next(4);
  observer.next(53);
  observer.complete();           // Fine emissione dati
});
numeriRandom$.subscribe(val => console.log("Dati: " + val));
```

Grazie al callback `next()` siamo in grado di emettere dei valori o eventi in modalità push. Nel nostro caso, emettiamo in successione il valore 4 e 53 per poi terminare richiamando il metodo `complete()`. Il relativo diagramma di Marble è mostrato qui sotto:

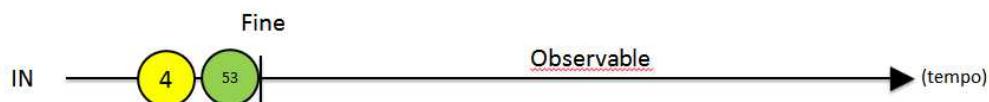


Figura 14.6 – Observable con emissione di due valori e successivo completamento

Come sempre per poter intercettare questi dati è necessario iscriversi. Grazie a quest'ultima notazione iniziamo a intuire un'altra importante caratteristica che avevamo accennato solo a livello di nome in precedenza.

Se ti ricordi avevamo parlato di Observable “Unicast” o “Cold” e avevamo riportato l'esempio del video su YouTube. Osservando con attenzione il codice notiamo che i valori emessi dall'Observable sono strettamente connessi a quest'ultimo. Nel senso che se effettuassimo due diversi `subscribe()` a distanza di qualche secondo, entrambi gli iscritti vedrebbero gli stessi dati dall'inizio.

```
numeriRandom$ = Observable.create(observer => {
    observer.next(4);
    observer.next(53);
    observer.complete();           // Fine emissione dati
});

numeriRandom$.subscribe(val => console.log("Datils: " + val));

setTimeOut(() => {
    numeriRandom$.subscribe(val => console.log("Dati2s: " + val));
}, 5000);
```

Potrebbe sembrare un'affermazione ovvia, se pensiamo a come è costruita ed eseguita una funzione, ma vedremo più avanti che non sempre questa caratteristica è quella che va bene per le nostre applicazioni. Avremo bisogno di creare degli Observable che abbiano un comportamento diverso.

Questo legame univoco tra sorgente e iscritto, viene chiamato “Unicast” ed è il comportamento predefinito di ogni Observable che creerai con i metodi visti.

Ogni iscritto vedrà sempre gli stessi dati un po' come accade con i video gratuiti registrati su YouTube.

Chiaramente lo stesso risultato ottenuto con il metodo `create()`, lo possiamo ottenere sfruttando direttamente il costruttore:

```
numeriRandom$ = new Observable(observer => {
    observer.next(4);
    observer.next(53);
    observer.complete();           // Fine emissione dati
});
```

In Angular capiterà raramente di creare un Observable con le tecniche viste all'inizio della sezione, ma sarà più probabile che sfrutterai i metodi che vedremo nelle prossime sezioni, dopo aver parlato di Observable Cold e Hot.

14.4 Cold vs Hot Observable

Prima di analizzare le caratteristiche dei più comuni operatori RxJS che userai nelle applicazioni Angular, è fondamentale approfondire il concetto di Observable Cold e Hot, senza il quale farai fatica a capire a fondo altri elementi chiave della libreria RxJS.

Devo essere sincero, la prima volta che ho letto questi termini sono rimasto un po' perplesso. Mi sono chiesto cosa c'entrasse il caldo e il freddo. Poi ho scoperto la reale natura di questi termini ossia Observable Unicast e Multicast.

Facciamoci aiutare dunque da qualche esempio della vita quotidiana. Qui di sotto ho elencato una serie di sorgenti d'informazioni che possono entrare rispettivamente nella categoria "Unicast" e "Multicast" o se preferisci "Cold" e "Hot".

- CD (Unicast) e Radio (Multicast)
- Film su cassetta (U) e TV (M)
- Video On Demand (U) e Video in Streaming (M)

Concentrandoci sul terzo esempio. Quando un utente acquista un video "On demand", può vedere il film dall'inizio, anche dopo 3 giorni dall'acquisto. La sorgente, ossia i server che gestiscono l'accesso al video, forniscono lo stesso video dall'inizio. Questo per ogni iscritto che acquista. Si crea un legame stretto tra chi produce i dati e chi li "consuma".

Un legame detto "unicast" perché ogni singolo utente ha la propria sorgente di dati che trasmette all'occorrenza. Le stesse considerazioni varranno chiaramente anche se il video è gratuito.

Quando un utente acquista il servizio di "Video Streaming", la sorgente diventa unica per tutti, in quanto si tratta di un evento in diretta. Se un utente si iscrive dopo che la diretta streaming è iniziata, non vedrà il video dall'inizio, come accade con un video "on-demand", ma dal punto in cui è giunta la trasmissione in quel momento.

Si crea così un legame "multicast" tra la sorgente e gli iscritti, perché la trasmissione del dato è rivolta contemporaneamente a più iscritti.

Quando definiamo un Observable con i metodi visti in precedenza, creiamo sempre un Observable Unicast. Questo significa che chiunque si iscriva, vedrà sempre gli stessi dati dall'inizio, un po' come accade quando noleggiamo un film.

Vediamo di tradurre questi concetti con qualche esempio di codice per confermare sul campo quanto detto. Nell'esempio qui sotto definiamo un Observable `myFilm`, che emette dei valori numerici ad intervalli regolari di due secondi, partendo dal numero uno. Potrebbe essere paragonato all'invio dei vari frame di un video.

```
this.myFilm$ = new Observable<number>(subscriber => {
    let num = 0;
    console.log('***** Si è iscritto un nuovo abbonato');
    setInterval(() => {
        num = num + 1;
        subscriber.next(num);
    }, 1000)
}) ;
...
```

Definiamo ora due funzioni che simuleranno l'iscrizione al servizio d'invio dei numeri, il nostro Observable `myFilm$`. Potrebbero essere paragonate a iscrizioni fatte su componenti diversi, una volta che l'Observable è centralizzato in un Service.

```
...
primoAbbonato() {
    this.abbonato1 = this.myFilm$.subscribe(num =>
        console.log("1° Abbonato: " + num)
    );
}

secondoAbbonato() {
    this.abbonato2 = this.myFilm$.subscribe(num =>
        console.log("2° Abbonato: " + num)
    );
}
...
```

Come puoi notare, all'interno di ogni funzione abbiamo inserito l'operazione d'iscrizione. Senza questa, come ricorderai, nessun dato potrà essere visualizzato.

Non appena richiameremo la prima funzione “`primoAbbonato`” avverrà l'iscrizione all'observable `myFilm$` che inizierà ad emettere i numeri in successione. Fino a qui nulla di strano.

Cosa succede se effettuo la chiamata alla seconda funzione “secondoAbbonato” dopo 5 secondi dalla prima? Vedrò i numeri dal valore 5 in poi o dall’inizio?

```
ngOnInit() {
  this.primoAbbonato();
  // mi abbono dopo 5 secondi
  setTimeout(() => {
    this.secondoAbbonato();
  }, 5000);
}
```

Questo quello che visualizzeremo nella console del browser sarà:

```
***** Si è iscritto un nuovo abbonato
Angular is running in the development mode. Call enableProdMode()
1° Abbonato: 1
1° Abbonato: 2
1° Abbonato: 3
1° Abbonato: 4
***** Si è iscritto un nuovo abbonato
1° Abbonato: 5
2° Abbonato: 1 ←→
```

Figura 14.7 – Valore visualizzato dal secondo iscritto all’Observable Unicast

L’ultima riga mostra che anche il secondo abbonato, vedrà la sequenza emessa dall’Observable partendo dal valore uno.

Se ricordiamo che un Observable non è altro che una funzione, il comportamento sicuramente non ci stupisce, ma in realtà c’è qualcosa di più profondo.

Se infatti spostiamo all’esterno dell’Observable la generazione del dato, otteniamo un comportamento diverso, che simula quello che accade quando ci iscriviamo ad un servizio di video in diretta streaming.

```
num = 0;
// cambio il valore ogni secondo
setInterval(() => {
  this.num = this.num + 1;
}, 1000);
```

(continua)

```

this.myObservable = new Observable<number>(subscriber => {
  console.log('***** Si è iscritto un nuovo abbonato');
  setInterval(() => {
    //num = num + 1;
    subscriber.next(this.num);
  }, 1000)
});

```

A seconda del momento in cui mi iscrivo, vedrò un dato diverso e non più la stessa sequenza di dati che vedevo prima. Mantenendo sempre la chiamata delle due funzioni primoAbbonato e secondoAbbonato distanziate di 5 secondi, quello che visualizzeremo nella console del browser sarà:

```

***** Si è iscritto un nuovo abbonato
Angular is running in the development mode. Call enablePr
1° Abbonato: 1
1° Abbonato: 2
1° Abbonato: 3
1° Abbonato: 4
***** Si è iscritto un nuovo abbonato
1° Abbonato: 5
2° Abbonato: 6 ←

```



Figura 14.7 – Valore visualizzato dal secondo iscritto all’Observable Multicast

L’ultima riga ci mostra chiaramente la differenza rispetto a prima, perché il secondo abbonato non visualizzerà più il numero 1 ma il numero 6, ossia il valore emesso dall’Observable dopo 6 secondi e il primo intercettato dal nuovo abbonato.

In questi casi si parla di Observable Multicast, ossia la sorgente non è collegata singolarmente all’iscritto (Unicast) ma contemporaneamente a più iscritti. La caratteristica peculiare quindi è che tutti coloro che si iscrivono, iniziano a vedere i dati da quel momento in poi, perdendo tutti i precedenti.

Puoi pensarlo come un singolo oratore che parla al microfono in una stanza piena di persone. Il loro messaggio - l’oggetto - viene consegnato a molte persone (gli osservatori) contemporaneamente .

Abbiamo già detto che in Angular, la classe HttpClient con i propri metodi, fornisce sempre Observable Unicast, così come l'interfaccia ActivatedRoute con le proprietà params, queryParams, fragment, data.

La domanda allora che potresti farti è in quali situazioni ci potrebbe essere utile creare un Observable Multicast (Hot) e quali metodi la libreria RxJS mette a disposizione per farlo. Tutto questo lo vedremo nelle prossime sezioni.

14.5 Operatori pipe(),map(),tap(),filter()

Arrivati a questo punto del capitolo, abbiamo esplorato gran parte delle peculiarità di un Observable che ci hanno permesso di capire più a fondo il significato delle righe usate nei capitoli precedenti. Ora abbiamo tutte le basi per iniziare a esplorare il capitolo legato alla manipolazione dei dati/eventi emessi dagli Observable.

Iniziamo dunque a parlare di operatori RxJS. In realtà possiamo dire già di conoscerli qualcuno perché fromEvent(), from(), of() e subscribe() sono degli operatori. Un operatore alla fine non è altro che una funzione, quindi prende un dato in ingresso, lo elabora senza cambiarlo e lo restituisce. Nel nostro caso la peculiarità è che il dato in ingresso è un Observable.

Come tutti gli operatori, per essere usati nelle applicazioni Angular devono essere importati dalla libreria RxJS che si installa non appena crei un nuovo progetto. In ogni componente, service, modulo ecc, dovrà quindi esserci una riga simile a questa:

```
import { listaoperatori RxJS } from 'rxjs/operators';
```

I riferimenti alla libreria installata e alla versione li troverai all'interno del file *package.json* in corrispondenza alla proprietà dependencies.

```
dependencies": {
  ...
  "rxjs": "~X.X.X",
  ...
}
```

Come tutte le funzioni, anche gli operatori sono usati per estrarre o confezionare in modo migliori i dati. Dati che tipicamente arrivano in seguito a interrogazioni di API

su server remoti o in seguito a operazioni fatti dell'utente nell'interfaccia dell'applicazione.

Partiamo con il primo operatore che abbiamo incontrato nel Capitolo 12. Analogamente all'API JavaScript `map()` anche nella libreria RxJS esiste l'equivalente operatore `map()` che agisce su un Observable. L'esempio rappresentato con il diagramma di Marble seguente, è molto esplicito.

Come vedi è presente il classico Observable iniziale, rappresentativo della sorgente originaria. Poi sotto si inserisce un rettangolo con all'interno il nome dell'operatore e una rappresentazione di quale operazione esegue.

E infine il nuovo Observable che viene generato. Il tutto sempre in riferimento allo scorrere del tempo.

In particolare nell'esempio si evince che ogni dato in ingresso rappresentato da una serie di numeri, è trasformato prendendo ogni numero e moltiplicandolo per 10.

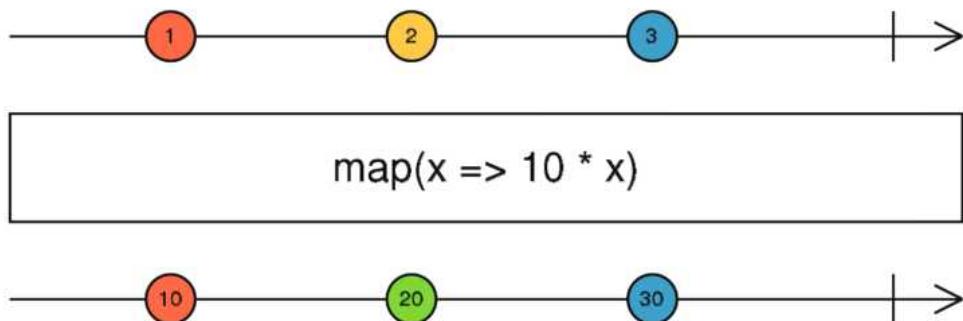


Figura 14.8 – Diagramma di Marble dell'operatore RxJS `map()`

Come già detto più volte, la cosa interessante da osservare è che il dato sorgente non viene modificato, e quello che otteniamo in uscita è un nuovo Observable.

Nell'applicazione Angular che abbiamo sviluppato, l'operatore `map()` è sempre stato inserito in tutte le chiamate all'API remota.

Ad esempio, se avessi anche sviluppato l'API per recuperare l'elenco degli iscritti all'applicazione MetroChat, in grado quindi di fornirmi per ogni utente interrogato un dato di questo tipo:

Lista-amici.json

```
[{
    "id": "1",
    "nome": "Delaney",
    "cognome": "Tonbridge",
    "email": "dtonbridge0@army.mil",
    "ip": "22.28.66.168",
    "amici": [
        {
            "id": "WRETY",
            "nome": "Colon Salazar",
            "citta": "Milano",
        },
        {
            "id": 2154,
            "nome": "French Mcneil",
            "citta": "Roma",
        },
        {
            "id": 3365,
            "nome": "Carol Martin",
            "citta": "Milano"
        }
    ]
}]
```

Se volessi recuperare solo la proprietà “amici” e quindi l’array di oggetti con proprietà `id`, `nome`, `citta`, con associato un tipo `Amico`, definito all’interno della cartella `model` con le tecniche che già conosciamo:

model/amico.model.ts

```
export interface Amico {
    id: string;          // identificativo dell'utente
    nome: string;        // nome dell'utente
    citta: string;       // url immagine avatar
}
```

potremmo usare l’operatore `map()` applicandolo all’`Observable` ottenuto interroghando l’API all’URL indicato in appendice al libro.

Creiamo un nuovo Service di nome *InfoUserService.ts* con all'interno il metodo `getInfoUser()`. Tale metodo interrogherà il server con una richiesta in GET quindi restituirà un Observable di tipo `Amico[]`.

```
service/info-user.service.ts

//...
import { map,tap,catchError } from 'rxjs/operators';

@Injectable({
  providerIn: 'root'
})
export class InfoUserService {
  private apiGetUrl = 'https://angular.dcopelli.it/amici/';
  constructor(private http: HttpClient) {}

  getInfoUser(idu:string): Observable<Amico[]> {
    return this.http.get<Amico[]>(this.apiGetUrl + '?idu=' + idu)
      .pipe(
        map((info:any) => info['amici']),
        catchError(this.handleErrorObs)
      );
  }
  ...
}
```

Grazie all'operatore `map()`, trasformo l'Observable ricevuto con `get()`, in un nuovo Observable, prelevando dall'originale solo la parte d'interesse ossia `info['amici']`.

Nota la presenza dell'operatore RxJS `pipe()` (pipe=tubo) che permette di raggruppare insieme più operatori, così come se stessimo costruendo un tubo e avessimo la necessità di collegare insieme vari pezzi.

```
.pipe(
  operatoreA(resp => resp1), // prima elaborazione
  operatoreB(resp1 => resp2), // seconda elaborazione
  ...
)
```

E' stato introdotto nella versione 5.5 di RxJS per poter gestire con meno codice la concatenazione di diversi operatori inseriti in successione, come avviene con i Promise grazie a `then()`.

Infatti essendo il dato restituito da `map()` un Observable, potrà a sua volta essere elaborato con altri operatori fino ad arrivare all'Observable finale.

Nel nostro esempio, il secondo operatore è `catchError()`, che gestirà l'invio dell'errore a colui che si iscriverà all'Observable.

Sappiamo infatti che fino a che non effettuiamo l'iscrizione all'Observable, questo non produrrà alcun dato. E' necessario che qualcuno lo attivi grazie all'operazione di `subscribe()`. Operazione che sarà effettuata all'interno di un componente in cui inietteremo il Service appena creato.

Concentrandosi solo sulla parte d'interesse del componente `info.component.ts` che creeremo appositamente per visualizzare tale lista, e recuperando il dato identificativo dell'utente direttamente dalla query nell'URL, scriveremo:

```

info/info.component.ts

// ...
import { Subscription } from 'rxjs';
@Component({
selector: 'ca-info',
template: `
<ca-menu></ca-menu>
<h3>Amici dell'utente con id: {{ idu }}</h3>
<div class="infouser">
<ul>
<li *ngFor="let amico of listaamici">
{{ amico.nome }}
</li>
</ul>
<div *ngIf="errormsg">{{errormsg}}</div>
</div>`})
export class InfoComponent implements OnInit, OnDestroy {
amiciSub!: Subscription;
listaamici!: Amico[];
idu!: string;
errormsg!: string;
constructor(private info: InfoUserService, private route: ActivatedRoute) {}
ngOnInit() {
this.idu = this.route.snapshot.paramMap.get('id')!;
this.amiciSub = this.info.getInfoUser(this.idu)
.subscribe(
amici => this.listaamici = amici,
error => this.errormsg = error,
() => console.log("Fine")) (continua)
}

```

```

        );
    }

ngOnDestroy() {
    this.amiciSub.destroy();
}
}

```

Abbiamo creato la variabile `amiciSub` di tipo `Subscriber` al fine di poter cancellare l'iscrizione all'`Observable` e liberare risorse. Poi abbiamo creato la variabile `listaamici` di tipo `Amico`, che useremo nel template del componente per mostrare la lista degli amici così recuperata.

Il dato associato all'identificativo dell'utente di cui vogliamo conoscere gli amici è recuperato tramite la query passata nell'url. Potremmo, ad esempio, dare la possibilità all'utente di cliccare sull'avatar presente in ogni messaggio chat (componente `ChatComponent.ts`) per accedere a questa nuova sezione dell'app.

```

// link alla route amici/:id
<a routerLink="/amici/{{msgIn.idu}}>
    
</a>

```

La direttiva `*ngFor` di Angular ci serve per scorrere l'elenco degli amici e per visualizzarli all'interno di una lista non ordinata. Il risultato che vedremo nel browser sarà simile a questo:



Figura 14.9 – Visualizzazione elenco amici utente con id=WRETY

Applicando invece le tecniche della programmazione reattiva, potremmo demandare l’iscrizione all’Observable direttamente nel template sfruttando il pipe `async`.

```
<li *ngFor="let amico of listaamici | async">
    {{ amico.nome }}
</li>
```

In quest’ultimo caso, nel corpo della classe la variabile `listaamici` dovrà essere definita come un Observable e non sarà più necessaria la variabile `amiciSub` per la cancellazione dell’iscrizione, in quanto questa avviene in automatico non appena il componente è eliminato:

```
//...
export class InfoComponent implements OnInit {
    ...
listaamici!: Observable<Amico[]>;
    ...
    ngOnInit() {
        this.idu = this.route.snapshot.paramMap.get('id')!;
        this.listaamici = this.info.getInfoUser(this.idu);
    }
    ...
}
```

Questo ci permetterà di scrivere un codice più pulito in quanto elimineremo le righe per la gestione dell’iscrizione. Dovremo però implementare una gestione degli errori che non faccia uso di una variabile valorizzata nel corpo della classe del componente, ma direttamente nel Service sfruttando le tecniche che vedremo nel prossimo capitolo.

Fare qualcosa con `tap()`

Tutte le volte in cui all’interno della sequenza degli operatori vogliamo controllare la correttezza dei dati elaborati con la classica istruzione di `console.log()`, o eseguire in modo imperativo un calcolo o assegnazione, ci possiamo far aiutare dall’operatore `tap()`, che può essere inserito in più punti del “tubo” di operatori che vado a concatenare.

Ad esempio nel caso del Service creato in precedenza potremmo scrivere:

service/info-user.service.ts

```
//...
@Injectable({
  providerIn: 'root'
})
export class InfoUserService {
  private apiUrl = 'https://angular.dcopelli.it/amici/';

  constructor(private http: HttpClient) {}

  getInfoUser(id:string, citta:string): Observable<Amici[]>{
    return this.http.get<Amici[]>(this.url + id)
      .pipe(
        tap(dato => console.log("Prima: " + dato)),
        map((info:any) => info['amici']),
        tap(dato => console.log("Dopo: " + dato)),
        catchError(this.handleErrorObs)
      );
  }
  ...
}
```

In questo caso mostro nella console del browser i dati in arrivo prima di eseguire la trasformazione con `map()` e dopo averli trasformati.

Filtrare i dati con filter()

Altro operatore che spesso si ha la necessità di utilizzare è quello che permette di filtrare dei dati di un Observable sulla base di una specifica condizione. E' molto simile all'equivalente operatore presente nella libreria JavaScript.

Spesso viene usato per filtrare i dati ricevuti in risposta ad una chiamata HTTP, oppure nei giochi online per filtrare i tasti della tastiera premuti da un giocatore, oppure per filtrare i dati inseriti in un campo di input di un form ecc.

Nel nostro caso lo potremmo usare per filtrare la lista degli utenti sulla base della città in cui vivono. Anche in questo caso il diagramma di Marble è di facile comprensione.

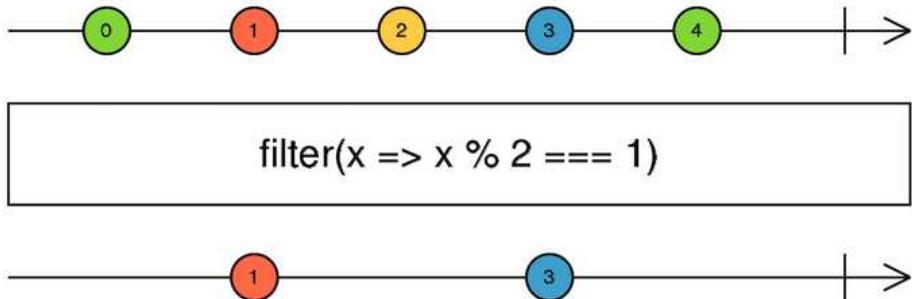


Figura 14.9 – Diagramma di Marble dell’operatore RxJS `filter()`

Quello che passo all’operatore `filter()` è una funzione che restituisce o il valore booleano `true` o il valore `false` a seconda di una condizione. Solo gli elementi che soddisfano alla condizione `true`, formeranno il nuovo Observable. Tutti gli altri saranno esclusi.

Ad esempio, se volessimo filtrare solo gli utenti della città di Milano, una possibile soluzione potrebbe essere quella di filtrare direttamente l’array associato alla proprietà `amici`, recuperato dopo l’applicazione del primo operatore `map()`. Sarà quindi sufficiente usare il classico operatore `filter` di JavaScript.

```
service/info-user.service.ts

//...
@Injectable({
  providerIn: 'root'
})
export class InfoUserService {
  private apiUrl = 'https://angular.dcopelli.it/amici/';
  constructor(private http: HttpClient) {}

  getInfoUser(id:string, citta:string): Observable<Amici[]> {
    return this.http.get<Amici[]>(this.url + id)
      .pipe(
        tap(dato => console.log("Prima: " + dato)),
        map((info:any) => info['amici'].filter(amico =>
          amico.citta == citta)),
        tap(dato => console.log("Dopo: " + dato)),
        catchError(this.handleErrorObs)
      );
  }
}
```

Quello che visualizzeremo nel browser , sarà la lista de i soli amici che vivono nella città di Milano.



```

 
Angular is running in the development mode. Call enableProdMode() to enable production mode.

Prima:
[{"id": "1", "nome": "Delaney", "cognome": "Tonbridge", "email": "dtonbridge0@army.mil", "ip": "22.28.66.168", "amici": [{"id": "2548", "nome": "Colon Salazar", "citta": "Milano"}, {"id": "2154", "nome": "French Mcneil", "citta": "Roma"}, {"id": "3365", "nome": "Carol Martin", "citta": "Milano"}]}

Dopo: [{"id": "2548", "nome": "Colon Salazar", "citta": "Milano"}, {"id": "3365", "nome": "Carol Martin", "citta": "Milano"}]

```

Figura 14.10 – Filtraggio dei dati sulla base della proprietà citta

Chiaramente tutti questi concetti valgono anche nel caso di eventi collegati al DOM. Giusto per preparare il campo per il successivo step di sviluppo dell'applicazione MetroChat che faccia uso di un Observable Hot, vediamo il caso in cui si voglia filtrare il tipo di tasti premuti dall'utente. Questo accade nei giochi, dove tipicamente sono ammessi solo specifici tasti legati a particolari operazioni.

Per generare un flusso di eventi del DOM che sia un Observable, possiamo sfruttare fromEvent e monitorare l'evento keydown. Quello che potremmo fornire in uscita è un flusso di oggetti del tipo:

```
`"spacebar": 32`
```

dove la prima informazione rappresenta il nome del tasto premuto e il valore, la codifica del tasto corrispondente.

Non tutti i tasti però hanno associato un nome quindi è necessario effettuare un primo filtro per intercettare solo i tasti che hanno un nome.

Tutti i tasti che non hanno un nome, saranno codificati come undefined e filtrati con l'operatore filter.

```

const keysDown$ = fromEvent(document, 'keydown').pipe(
    map((event: KeyboardEvent) => {
        const name = KeyUtil.codeToKey(''+event.keyCode);
        if (name !== '') { (continua)

```

```

        let keyMap = {};
        keyMap[name] = event.code;
        return keyMap;
    } else {
        return undefined;
    }
),
filter((keyMap) => keyMap !== undefined)
);

}

```

14.6 MultiCast Observable: Subject vs Behaviour Subject

Prima di entrare nel vivo dell'argomento, cerchiamo di rispondere alla domanda rimasta in sospesa nella precedente sezione, ossia perch é è necessario creare degli Observable Multicast.

Nelle applicazioni Angular spesso si ha la necessità di aggiornare o mantenere lo stato. Stato che può essere vincolato o a dati prelevati da un server remoto tramite un'API o a azioni fatte sull'interfaccia utente (UI).

Tipico esempio potrebbe essere un dato che rappresenta il totale articoli presenti nel database e mostrati come badge nel menu di tutte le pagine dell'app indipendentemente dalla sezione che l'utente sta navigando.

Altro esempio potrebbe essere il totale articoli inseriti in una shopping cart, o il numeri di treni in arrivo, come nel caso dell'applicazione MetroChat.

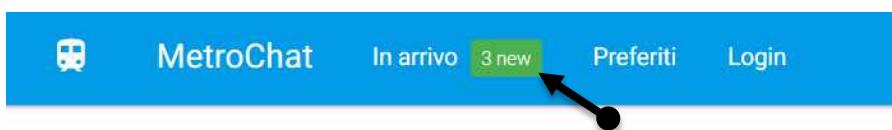


Figura 14.11 – Visualizzazione in tempo reale del numero di treni in arrivo

Questo significa che, a seconda delle funzionalità dell'applicazione, diversi componenti potenzialmente avranno la necessità di conoscere questo dato in tempo reale.

Invece di definire ogni dato di questo tipo all'interno del componente radice di tutti gli altri, e passarlo ai vari figli sfruttando il meccanismo di comunicazione tra componenti con dati di input e output, o tramite l'indirizzo url, o altre tecniche, potremmo definirlo come un Observable Multicast (Hot) centralizzato.

In questo modo tutti i componenti che si iscriveranno per ricevere i dati, anche se non sono strettamente collegati da una relazione genitore/figlio nell'albero dei componenti dell'applicazione, vedranno sempre l'ultimo valore emesso, che è il valore aggiornato dello stato, con un notevole risparmio sia di codice, sia di potenziali errori.

Se poi il tutto viene gestito all'interno di un Service, ogni componente potrà, non solo leggere il dato, ma anche aggiornarlo, tramite un'interfaccia che non esponga direttamente l'Observable.

E' come se creassimo una sorgente di dati globale accessibile in lettura e scrittura da tutti i componenti dell'applicazione, con il vantaggio che tutti gli aggiornamenti dello stato ci saranno comunicati non appena disponibili, in modo asincrono.

Non dobbiamo pertanto preoccuparci di decidere in che punto, di ogni componente, leggere e assegnare il nuovo stato, perché tutto questo avverrà grazie all'iscrizione ad un Observable e sfruttando il meccanismo del "change detection" per l'aggiornamento dell'UI.

Sebbene esistano diverse librerie dedicate alla gestione dello stato, come NgRx, Ngxs, MobX, Akita, ecc, si possono ottenere buoni risultati sfruttando ancora la libreria RxJS.

In particolare sfruttando gli Observable Subject e BehaviourSubject.

Anche in questo caso, la prima volta che ho letto questi termini sono rimasto un po' perplesso perché non sono riuscito a cogliere, nella traduzione letterale, alcun collegamento efficace con il mondo reale.

Chiariamo subito che entrambi ci permettono di definire un Observable, solo che a differenza di quelli che abbiamo visto fino ad ora, sono Observable Multicast, cioè in grado di aggiornare contemporaneamente più iscritti con lo stesso dato.

Molti usano il termine "EventEmitter" e in effetti se guardiamo la definizione della classe, scopriamo la reale natura:

```
class Event Emitter<T> extends Subject {
  constructor(isAsync: boolean = false)
  emit(value?: T)
  subscribe(generatorOrNext?: any, error?: any, complete?: any):
    Subscription
}
```

Per rifarci a qualche esempio del mondo reale, puoi pensarla come un singolo oratore che parla al microfono in una stanza piena di persone. Il suo messaggio (l'oggetto) viene consegnato a molte persone (multicast) (gli osservatori) contemporaneamente. Questa è la base del concetto di Multicast.

Una delle caratteristiche peculiari sia di `Subject` che di `BehaviourSubject` è che entrambi sono allo stesso tempo sia degli `Observable` che degli `Observer`, quindi sono in grado sia di emettere dei valori, sia di leggerli.

Vediamo allora come si possano definire all'interno di un'applicazione, così come fatto per i normali `Observable`.

La prima cosa, sarà importarli dalla relativa libreria RxJS e poi dovremo istanziare la relativa classe.

```
import { Subject, BehaviorSubject } from 'rxjs';

const subject$ = new Subject(); // Hot
const behaviourSubject$ = new BehaviourSubject(0) // Hot + valore
```

E' chiaro che presentano delle differenze, e la più evidente è che `BehaviourSubject` è in grado di emettere subito l'ultimo valore emesso, anche se nel frattempo è già stato emesso.

Questo significa che ogni nuovo iscritto all'`Observable`, vedrà subito un valore, senza attendere che ne venga emesso uno nuovo, come accade per tutti gli altri `Observable`.

Non demoralizzarti se ti sembra tutto un po' fumoso. E' normale che tu stia facendo fatica. Vedrai che rileggendo queste righe, dopo essere arrivato a fine capitolo, le cose diventeranno più chiare.

Non per mettere il dito nella piaga ma giusto per completezza, oltre a `BehaviorSubject` esistono altre due varianti di `Subject` che tipicamente sono usate meno spesso rispetto ai due precedenti. So parlando di `AsyncSubject` e `ReplaySubject`.

Il primo emette l'ultimo valore agli iscritti, una volta che l'`Observable` ha terminato di emettere valori. Il secondo emette agli iscritti gli ultimi N valori, dove N è un dato che è necessario specificare.

Giusto per fare un po' più di chiarezza, ho voluto riepilogare qui sotto le caratteristiche peculiari di ognuno in modo che tu abbia un rapido collegamento da consultare

- **Observable**: unicast, crea una copia dei dati
- **Subject**: multicast, condivide i dati, non emette alcun valore iniziale agli iscritti
- **BehaviorSubject**: multicast, condivide i dati, richiede un valore iniziale ed emette il valore corrente (ultimo elemento emesso) per i nuovi abbonati, anche dopo che è stato emesso
- **AsyncSubject**: multicast, condivide i dati, emette l'ultimo valore per gli iscritti al completamento.
- **ReplaySubject**: multicast, condivide i dati, emette un numero specifico di ultimi valori emessi (un replay) per i nuovi iscritti.

Si potrebbe pensare all'Observable Multicast come ad una sorta di memoria temporanea accessibile da tutti, che riflette in automatico il proprio aggiornamento a coloro che la stanno usando.

Qui sotto una tabella ne riassume alcune caratteristiche e differenze sostanziali rispetto agli Observable.

Observables	Subject	Replay Subject	Behavioral Subject
Cold	Hot	Hot	Hot
Creates copy of data	Shares data	Shares data	Shares data
Uni-directional	bi-directional	bi-directional	bi-directional
-	-	Replay the message stream	Replay the message stream
-	-	-	You can set initial value

Figura 14.12 – Elenco caratteristiche e differenza dei diversi tipi di Observable

Vediamo di scrivere qualche esempio di codice per confermare tutto quello detto. Proviamo a vedere cosa succede non appena ci iscriviamo ad entrambi gli Observable.

```
import { Subject, BehaviorSubject } from 'rxjs';
const subject$ = new Subject(); // Definisco un Observable Hot
// Definisco un Observable Hot con valore predefinito iniziale
const behaviourSubject$ = new BehaviourSubject(0);
```

(continua)

```

subject$.subscribe({
  next: (v) => console.log(`1° Abbonato : ${v}`)
});
behaviourSubject$.subscribe({
  next: (v) => console.log(`2° Abbonato: ${v}`)
});

```

Nella console del browser vedremo:



Figura 14.13 – Effetto dell’iscrizione a Subject e BehaviorSubject

Questo evidenzia come solo il secondo iscritto, quello a BehaviorSubject presenti subito un valore emesso, nonostante l’Observable non abbia ancora emesso alcun valore.

Proviamo allora a emettere dei valori, inserendo diverse chiamate al metodo `next()`, in punti diversi per simulare il trascorrere del tempo.

Concentriamo per ora solo su `Subject`. Emettiamo due va lori subito dopo l’iscrizione del primo abbonato, e vediamo se vengono intercettati.

```

import { Subject, BehaviorSubject } from 'rxjs';

const subject$ = new Subject(); // Definisco un Obsevable Hot

subject$.subscribe({
  next: (v) => console.log(`1° Abbonato a Subject : ${v}`)
});

subject$.next(1);
subject$.next(2);

subject$.subscribe({
  next: (v) => console.log(`2° Abbonato a Subject: ${v}`)
});

subject.next(3);                                (continua)

```

Quello che vedremo nella console del browser sarà:

```
1° Abbonato a Subject: 1
1° Abbonato a Subject: 2
1° Abbonato a Subject: 3
2° Abbonato a Subject: 3
```



Figura 14.14 – Valori emessi per due differenti iscrizioni ad un Observable Subject

Se fosse un normale Observable, come ricorderai, anche per il secondo Observable dovremmo visualizzare i valori 1,2 ossia tutti i valori emessi fino a quel momento . Quello che invece visualizziamo è il valore 3.

L'ultima riga ci dice inoltre che il 2° abbonato visualizzerà il nuovo valore solo dopo una nuova chiamata a `next()` e non all'atto dell'iscrizione come invece avviene se usiamo BehaviourSubject.

Infatti se ripetiamo lo stesso ragionamento anche per quest'ultimo, nota cosa succede nel momento in cui il secondo abbonato si iscrive.

```
import { Subject, BehaviorSubject } from 'rxjs';

const behaviourSubject$ = new BehaviorSubject(0);

behaviourSubject$.subscribe({
  next: (v) => console.log(`1° Iscritto a BehaviourSubject: ${v}`)
});

behaviourSubject$.next(1);
behaviourSubject$.next(2);

behaviourSubject$.subscribe({
  next: (v) => console.log(`2 Iscritto a BehaviourSubject$: ${v}`)
});

behaviourSubject$.next(3);
```

La freccia presente nell'immagine qui sotto, indica il momento in cui avviene l'iscrizione. E' subito visualizzato un valore, corrispondente all'ultimo dato emesso, ossia 2.

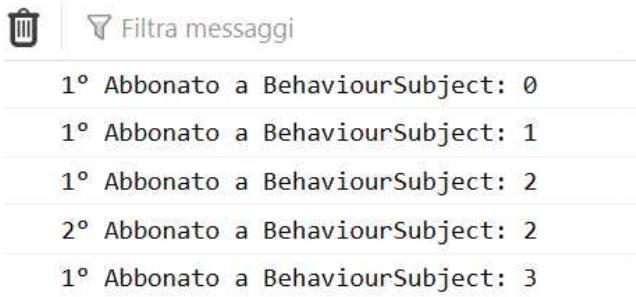


Figura 14.15 – Valori emessi per due differenti iscrizioni ad un Observable BehaviorSubject

Chiaramente non ti troverai mai nella situazione di dover inserire più chiamate al metodo `next()` per generare nuovi valori, ma questi tipicamente saranno emessi in seguito a chiamate multiple via http ad un’API, o tramite un websocket o tramite modifiche dirette nell’UI.

Inoltre tipicamente questi Observable sono gestiti all’interno di un Service al fine di renderli disponibili in più punti dell’applicazione.

L’ultima caratteristica importante di `BehaviourSubject` che lo contraddistingue da `Subject`, è che posiamo in ogni istante recuperare l’ultimo valore emesso dall’Observable sfruttando il metodo `getValue()`.

```
let lastValue = behaviourSubject$.getValue();
```

Ora che abbiamo capito più a fondo la differenza tra i due Observable, vediamo come sfruttarli riportando un ultimo esempio che sia più vicino alle comuni applicazioni che svilupperai.

Nell’ipotesi tu gestisca un sito di e-commerce con tre prodotti e voglia creare una dashboard per conoscere in tempo reale i ricavi via via che gli ordini arrivano, potresti interrogare, all’atto della creazione dell’istanza dell’app, il tuo server e recuperare così il dato.

Una situazione simile a quella dell’applicazione MetroChat, dove in corrispondenza all’arrivo di nuovi treni, mostro un timer con il conteggio del tempo rimasto alla partenza.

In questo esempio, a seguito dei nuovi ordini realizzati nel sito, si creerebbe un nuovo ricavo, quindi un nuovo totale, che potrebbe essere gestito con un Observable Multicast a cui tutti i componenti potranno iscriversi, per conoscere in tempo reale i guadagni della giornata.

Ipotizzando di aver già popolato un array con un elenco di possibili ordini:

```
ordini = [
  {id: "123ED", prodotto: "Pasta", prezzo: 1.20, qta: 4},
  {id: "233XC", prodotto: "Pomodoro", prezzo: 1.50, , qta: 12},
  {id: "233XC", prodotto: "Mozzarella", prezzo: 4.50, , qta: 2}
];
```

potremmo simulare l'arrivo di questi nel tempo, in vari modi, ad esempio con una funzione classica JavaScript `setInterval()` che va a richiamare il metodo `newOrdine()` ogni 3 secondi.

```
...
ordiniSubject$ = new Subject();
interval: number;

constructor() {
  this.interval = setInterval(() => this.newOrdine(), 3000);
}

...
```

Tale metodo andrà a prelevare sequenzialmente i diversi elementi dell'array e li emetterà tramite un Observable `Subject` di nome `subjectDati`.

```
...
newOrdine() {
  if (this.num < this.ordini.length) {
    this.ordiniSubject$.next(this.ordini[this.num]);
    this.num++;
  } else {
    this.ordiniSubject$.complete();
    clearInterval(this.interval);
  }
}
```

Arrivati alla fine dell'array, l'Observable terminerà l'invio dei dati richiamando il metodo `complete()`. Abbiamo creato un rudimentale simulatore di ordini in arrivo, ma che può essere la base per molti esempi di codice che dovrà sviluppare in cui uno stesso dato asincrono deve poter essere letto da più componenti

contemporaneamente (es. conteggio numero treni in arrivo per l'applicazione MetroChat)

Trasformando que sti pezzi in un Service di nome *OrdiniService.ts* che inietteremo all'occorrenza a tutti i componenti che lo richiederanno, otterremo:

```
service/ordini.service.ts
```

```
import { Subject, BehaviorSubject } from 'rxjs';
import { Injectable } from '@angular/core';

@Injectable({
  providedIn: 'root'
})
export class OrdiniService {
  num: number = 0;
  ordini = [
    {id: "123ED", prodotto: "Pasta", prezzo: 1.20, qta: 4},
    {id: "233XC", prodotto: "Pomodoro", prezzo: 1.50, , qta: 12},
    {id: "233XC", prodotto: "Mozzarella", prezzo: 4.50, , qta: 2}
  ];

  ordiniSubject$ = new Subject();
  interval: number;

  constructor() {
    this.interval = setInterval(() => this.newOrdine(), 3000);
  }

  newOrdine() {
    if (this.num < this.ordini.length) {
      this.ordiniSubject$.next(this.ordini[this.num]);
      this.num++;
    } else {
      this.ordiniSubject$.complete();
      clearInterval(this.interval);
    }
  }
}
```

Questo service potrà ora essere sfruttato da ogni componente indipendentemente da dove si trova nell'albero dei componenti.

Se ipotizziamo che l'app sia costituita da un template in cui vogliamo mostrare un testo con il nome del prodotto associato all'ordine in arrivo, l'unica operazione che ci rimane da fare è iscriverci all'Observable:

dash.component.ts

```

import { OrdiniService } from './service/dati.service';
import { Component, OnInit, OnDestroy } from '@angular/core';
import { Subscription, Subject, BehaviorSubject } from 'rxjs';

@Component({
  selector: 'app-root',
  template: '<h2>App (1° subscribe): Prod.ricevuto: {{ prodotto }}</h2>'
})

export class DashComponent implements OnInit, OnDestroy {

  obsub: Subscription;
  prodotto: string;
  constructor(private datiservice: OrdiniService) {}

  ngOnInit() {
    this.obsub = this.datiservice.ordiniSubject$.subscribe(num => {
      this.prodotto = num['prodotto'];
    });
  }

  ngOnDestroy() {
    this.obsub.unsubscribe();
  }
}

```

Abbiamo iniettato il Service nel costruttore, poi all'interno di `ngOnInit()`, ci siamo iscritti all'Observable e infine abbiamo valorizzato la variabile `temp` di nome `prodotto`.

Tale variabile è proprio quella presente nel template del componente `dash.component.ts`, a cui assegniamo il valore presente nella proprietà `prodotto` `num['prodotto']`; dell'oggetto emesso ogni 3 secondi dall'Observable `ordiniSubject$`.

Ora che sappiamo le caratteristiche più profonde dell'Observable `Subject`, all'apertura della pagina, quello che visualizzeremo sarà una scritta senza indicazione del prodotto:

App (1° subscribe): Prodotto ricevuto:

Figura 14.16 – Messaggio visualizzato all’apertura della pagina in seguito all’iscrizione di un Subject

Al contrario di `BehaviorSubject`, dobbiamo attendere l’emissione di un valore, che avviene dopo 3 secondi. Infatti dopo tale periodo apparirà il valore “Pasta” seguito dopo 3 secondi da “Pomodoro” e così via.

App (1° subscribe): Prodotto ricevuto: Pasta

App (1° subscribe): Prodotto ricevuto: Mozzarella

Figura 14.17 – Messaggi visualizzati dopo 3 e 6 secondi dall’apertura della pagina

E’ chiaro che tutti i nuovi iscritti all’`Observable` vedranno sempre il successivo valore emesso dall’`Observable` `ordiniSubject$`.

Al fine di simulare , sullo stesso componente, un’ipotetica iscrizione fatta su un altro componente, potremmo inserire una nuova dopo 4 sec. la precedente:

```

dash.component.ts
// ...
@Component({
  selector: 'app-root',
  template: '<h2>App (1° subscribe): Prod. ricevuto: {{ prodotto }}</h2>
            <h2>App (2° subscribe): Prodotto ricevuto: {{ prodotto1 }}</h2>'
})
export class DashComponent implements OnInit, OnDestroy {

  obsub: Subscription;
  obsub1: Subscription;
  prodotto: string;
  prodotto1: string;

  constructor(private datiservice: OrdiniService) {}

  ngOnInit() {
    this.obsub = this.datiservice.ordiniSubject$.subscribe(num => {
      this.prodotto = num['prodotto'];
    });
  }
}
(continua)
```

```

setInterval(() => {
  this.obsub1 = this.datiservice.ordiniSubject$.subscribe(num => {
    this.prodotto1 = num['prodotto'];
  });
}, 4000);
}

```

Questo secondo iscritto inizialmente non visualizzerà nulla, perché l'ultimo valore disponibile “Pasta” è stato emesso allo scattare dei 3 secondi, quindi prima di questa nuova iscrizione.

App (1° subscribe): Prodotto ricevuto: Pasta

App (2° subscribe): Prodotto ricevuto:

Figura 14.18 – Messaggi visualizzati dopo 3 secondi dall’apertura della pagina

Al secondo invio, ossia dopo 6 secondi invece, entrambi gli iscritti visualizzeranno il nuovo valore:

Questo, ripeto, indipendentemente dal fatto che l’iscrizione sia avvenuto nello stesso componente. Lo puoi verificare creando un nuovo componente e inserendolo nello stesso template dell’applicazione, come richiesta dall’esercitazione che trovi in fondo al capitolo.

Nel caso invece usassi un Observable ottenuto con `BehaviorSubject`, l’unica differenza sarà sul valore emesso all’atto dell’iscrizione.

In assenza di dati emessi, sarà visualizzato il dato predefinito passato come parametro in fase di creazione dell’Observable, mentre se ci iscriviamo dopo che è stato emesso un valore, sarà visualizzato subito l’ultimo dato emesso.

Questo può essere utile quando ad esempio dobbiamo creare un’interfaccia che necessita di visualizzare subito un valore iniziale, ad esempio un totale o una quantità e così via.

Il codice così creato funziona, ma presenta una problematica che non dobbiamo trascurare, e che in applicazioni più complesse può causare diversi problemi : sto parlando dell’unicità della sorgente dati.

Avendo definito la variabile `ordiniSubject$` come membro pubblico della classe Service, chiunque può accedervi. Questo significa mettere a rischio l'integrità dell'applicazione perché potremmo inavvertitamente sovrascrivere la variabile o emettere dei valori non previsti. Ogni componente infatti può richiamare il metodo `next()` su `ordiniSubject$`.

E' buona norma trasformare quest'ultima in una variabile privata e non dare a nessuno la possibilità di emettere dati per conto suo.

Ma se la variabile diventa privata, nessuno dall'esterno della classe potrà accedervi. Questo ci costringe a usare un altro metodo della libreria RxJS, ossia `asObservable()`.

```
// trasforma il Subject in membro privato della classe
private ordiniSubject$ = new Subject();

dati$ = ordiniSubject$.asObservable();
```

Ho creato la variabile pubblica `dati$` e l'ho trasformata nel tipo Observable. Così facendo l'unica operazione che potremmo fare sarà l'iscrizione all'Observable `dati$` che emetterà i valori per conto di `ordiniSubject$`, senza andare incontro ai rischi descritti, in quanto un Observable, rispetto a `Subject`, non ha i metodi `next()`, `err()` e `complete()`.

Il codice del service pertanto diventerà:

```
service/ordini.service.ts

import { Subject, BehaviorSubject } from 'rxjs';
import { Injectable } from '@angular/core';

@Injectable({
  providedIn: 'root'
})
export class DatiService {
  num: number = 0;
  ordini = [
    {id: "123ED", prodotto: "Pasta", prezzo: 1.20, qta: 4},
    {id: "233XC", prodotto: "Pomodoro", prezzo: 1.50, , qta: 12},
    {id: "233XC", prodotto: "Mozzarella", prezzo: 4.50, , qta: 2}
  ];

  private ordiniSubject = new Subject();
  dati$ = ordiniSubject.asObservable();
  interval: number;
```

(continua)

```

constructor() {
    this.interval = setInterval(() => this.newOrdine(), 3000);
}

newOrdine() {
    if (this.num < this.ordini.length) {
        this.subjectDati.next(this.ordini[this.num]);
        this.num++;
    } else {
        this.subjectDati.complete();
        clearInterval(this.interval);
    }
}
}

```

Ogni componente interessato a ricevere i dati dall'Observable, potrà iscriversi direttamente a dati\$:

```

this.obsSub1 = this.datiservice.dati$.subscribe(num => {
    this.prodotto = num['prodotto'];
});

```

Esercitazione di verifica: prova a creare un nuovo componente testComponent .ts e ad iscriverti allo stesso observable ordiniSubject\$ del Service OrdineService.ts , per mostrare la scritta “Test (1° Subscribe): Prodotto ricevuto: ...” .

Prova poi a inserire il componente all'interno del template dell'app principale e a vedere se, in corrispondenza all'aggiornamento del dato, anche il componente si aggiorna contemporaneamente a quello principale.

14.7 Combinare flussi di Observable

Fino ad ora abbiamo sempre trattato un singolo flusso di Observable, ma in molte situazioni dovremo gestire più flussi di Observable.

Se ad esempio avessimo un Observable in grado di emettere un dato che è a sua volta un Observable, come possiamo gestirli senza impazzire in multipli callback?

Partiamo, come sempre, da un esempio riferito a una situazione della vita quotidiana per poi agganciarlo a una situazione che ti potrebbe capitare durante lo sviluppo di un'applicazione.

Operatore FlatMap()

Ipotizziamo di essere il titolare di un importante ristorante in centro a Milano e di avere due importanti ceremonie da gestire con più di 500 invitati ciascuno. Una situazione simile l'ho vissuta di persona quando lavoravo come "lavapiatti" a Londra nel secolo scorso.

Come responsabile del ristorante è chiaro che dobbiamo dare "ascolto" a tutti gli invitati di entrambe le ceremonie. Il primo problema che sorge è che gli invitati arrivano in ordine sparso. E' necessario quindi intercettarli in anticipo per smistarli nelle giuste sale.

Detto nel gergo che ormai conosciamo, dobbiamo "iscriversi" innanzitutto al flusso che emette il dato relativo all'arrivo degli invitati.

Non è però finita qui. Ad ogni invitato possiamo concedere la possibilità di fare delle richieste, quindi è necessario mettersi in ascolto anche delle loro richieste. Devo pertanto iscrivermi a questo secondo flusso di dati.

Probabilmente se vivi in una famiglia con più bambini, conosci molto bene queste dinamiche.

La situazione che si genera quindi è che dobbiamo iscriverci a più flussi di informazioni che arrivano in modo del tutto casuale. Creare però più iscrizioni, diventa difficile da gestire a livello di codice, perché iniziamo ad avere callback multipli nidificati.

Se invece avessimo un sistema che ci permetesse di trasformare questi due flussi in un unico flusso, riusciremmo a intercettare tutte le richieste e a gestirle in modo più efficiente.

Nel gergo dell'algebra lineare, potremmo dire che passiamo da una matrice, costituita da più righe e colonne ad un vettore. Un po' quello che succede quando sei in autostrada e da tre corsie si passa a una corsia. Il travaso dei flussi di ogni corsia avviene in modo graduale partendo da quella più esterna, un po' come l'arrivo dei gruppi di invitati. Ogni corsia genera un flusso di macchine indipendente, che viene trasformato in un unico flusso dal rest ringimento.

Per ottenere questa trasformazione abbiamo a disposizione un operatore della libreria RxJS, che prende appunto il nome di `flatMap()`.

La parola "Flat", nel nostro caso, ha il significato di appiattire o serializzare i dati.

Il diagramma di Marb le che troviamo nel sito ufficiale, permette di capire meglio questo concetto.

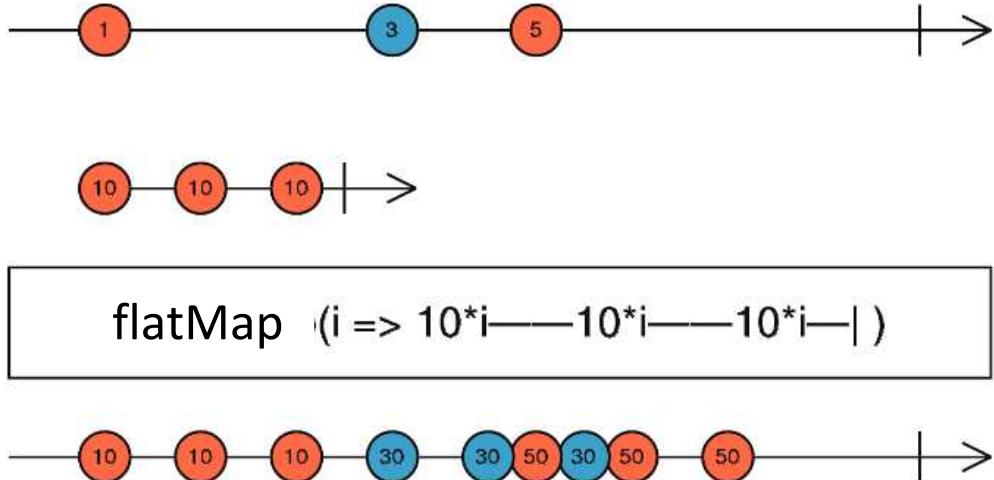


Figura 14.19 – Diagramma di Marble dell’operatore flatMap()

Al flusso esterno (l’arrivo dei diversi gruppi, nella nostra analogia), detto “Outer Observable”, si aggiungono una serie di flussi interni (singoli invitati), detto “Inner Observables”, che grazie all’operatore `flatMap()` vengono trasformati e poi combinati insieme in un nuovo singolo Observable, costituito dalla fusione del flusso esterno con tutti i flussi interni .

La fusione , mantiene la completa casualità di emissione dei vari dati a differenza di altri operatori che concatenano i diversi flussi.

Con una sola iscrizione all’Observable così generato da `flatMap()`, posso così intercettare tutte le richieste dei vari invitati a seconda dell’ordine di arrivo.

Traducendo tutto questo nelle applicazioni Angular, un tipico uso dell’operatore `flatMap()` è quando, per ogni richiesta via HTTP, si generano altre richieste, che richiedono il risultato della prima.

Sfruttando l’esempio visto nel capitolo precedente, per ogni ordine in arrivo potremmo effettuare la chiamata ad un’api, per recuperare la giacenza di magazzino tra tutti i magazzini della zona da cui arriva l’ordine, al fine di verificare se l’ordine può essere evaso oppure no.

Se non avessi a disposizione questo operatore, dovrei concatenare più subscribe .

```
this.dati.ordiniSubject$  
    .subscribe(num => {  
        this.http.get('/giacenza/' + num['id'])  
            .subscribe(res => this.rim = res)  
    });
```

Questo produce un codice , che se ripetuto per N chiamate, diventa poco leggibile e ritorniamo a d avere gli stessi problemi visti all'inizio, ossia un proliferazione di callback.

Sfruttando invece flatMap() o una combinazione di tanti flatMap(), possiamo usare un unico subscribe().

Come sempre per usarlo lo dobbiamo importare dalla relativa libreria. Il codice precedente potrebbe essere scritto in questo modo:

```
import { flatMap } from 'rxjs/operators';  
...  
  
this.dati.ordiniSubject$.pipe(  
    flatMap(num => this.http.get('/giacenza/' + num['id']))  
).subscribe(res => this.rim = res);
```

L'azione di subscribe, dell'Observable più esterno, fa partire in automatico anche il subscribe per tutti gli Observable interni, garantendo l'intercettazione di tutti i dati e non solo quelli dell'Observable più esterno.

Operatore SwitchMap()

Sempre nell'ambito della trasformazione di flussi di più Observable, l'operatore switchMap() presenta una peculiarità che lo distingue in modo inequivocabile dal precedente flatMap(). Questa caratteristica la possiamo già intuire dal nome.

Con switchMap() infatti, possiamo commutare (switch) ad un altro Observable interno, cancellando l'iscrizione al precedente.

Detto così, forse non ti è molto chiaro . Proviamo a sfruttare nuovamente l'esempio precedente. Cosa succede se un ospite della prima cerimonia chiede al gestore del ristorante di portagli un bicchiere di vino e dopo 10 secondi un altro invitato dello

stesso banchetto , arrivasse al banco del ges tore per annullare la precedente richiesta a favore di un bottiglione da 3 litri?

Il gestore del ristorante sicuramente lo guarderebbe con occhi spalancati, ma gli ospiti sono ospiti, quindi dovrebbe subito segnalare al cameriere di bloccare la precedente richiesta e ascoltare questo nuovo ospite , al fine di portare il bottiglione.

Non avrebbe senso infatti portare al tavolo il bicchiere e poi tornare indietro per portare il bottiglione. Il numero di camerieri di un ristorante è sempre limitato al massimo, per massimizzare i profitti, quindi devono essere il più possibile produttivi. L'operatore `switchMap()` è in un certo senso ottimizza questa gestione.

Lo possiamo vedere meglio analizzando il relativo diagramma Marble:

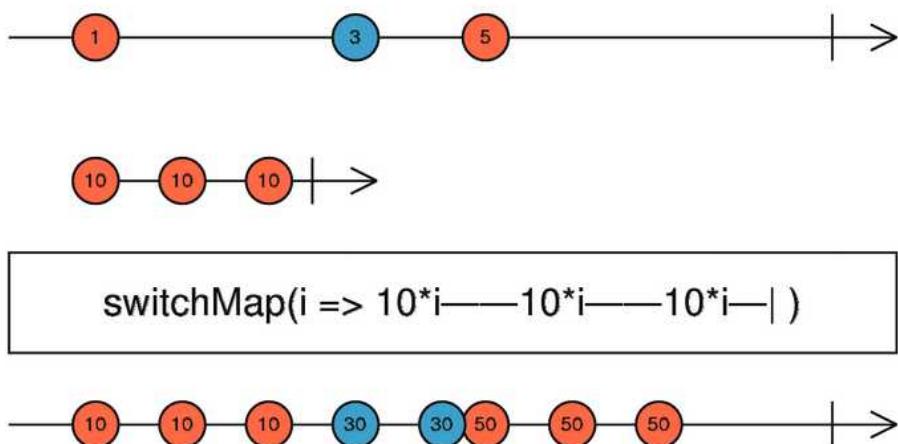


Figura 14.20 – Diagramma di Marble dell'operatore `switchMap()`

La cosa interessante è che l'operazione di cancellazione è fatta in automatico, non appena arriva un nuovo flusso dall'Observable più esterno. Non dobbiamo pertanto preoccuparci di annullare alcuna iscrizione .

Nelle applicazioni Angular, ci possono essere diverse situazioni in cui questo comportamento è necessario, ma le più comuni che affronterai sono l'implementazione di un classico campo di ricerca che interroga un database remoto tramite un'API, oppure il recupero in successione dei parametri di un URL legati a cambi di pagina repentini o alle richieste del dettaglio di più articoli di una pagina prodotti.

Nel primo caso la digitazione dei tasti in successione , rappresenta un flusso di eventi “keyup” associati a una particolare lettera generata, che a sua volta genera una richiesta ad un server remoto che recupera un nuovo dato. Questi due flussi devono essere combinati insieme.

Fino a qui potresti pensare che l’operatore `flatMap()` sia sufficiente per gestire il tutto, ma non sarebbe efficiente.

Quando digitiamo delle lettere nel campo di ricerca di un’applicazione, produciamo una sequenza di eventi legati alla pressione dei tasti. Ognuno di questi genera potenzialmente una richiesta all’API legata al server remoto dove sono memorizzati i dati da cercare .

Questo implica un’enorme quantità di richieste e di tempo d’attesa. La maggior parte delle quali può essere evitato perché spesso è l’intera parola che si vuole cercare e non le singole lettere o gruppi di lettere.

Grazie all’operatore `switchMap()`, ad ogni nuovo evento legato alla pressione di un tasto, eliminiamo la precedente richiesta non ancora evasa, e ci iscriviamo alla nuova richiesta.

Questo ci garantisce un primo filtro sul tempo d’attesa, potendoci concentrare solo sull’ultima richiesta basata sulla sequenza completa di lettere inserite .

Anche in questo caso l’operatore deve essere importato dalla relativa libreria:

```
import { switchMap, debounceTime, catchError } from 'rxjs/operators';
```

Proviamo ad applicare questi concetti all’applicazione MetroChat, realizzando un campo di ricerca che ci permetta di filtrare l’elenco dei messaggi visualizzati non appena entriamo nel dettaglio delle chat di ogni treno.

Aggiungiamo un campo di ricerca nel template del componente `dettaglio.component.ts` e creiamo il relativo form per intercettare ogni inserimento di dati effettuato dall’utente.

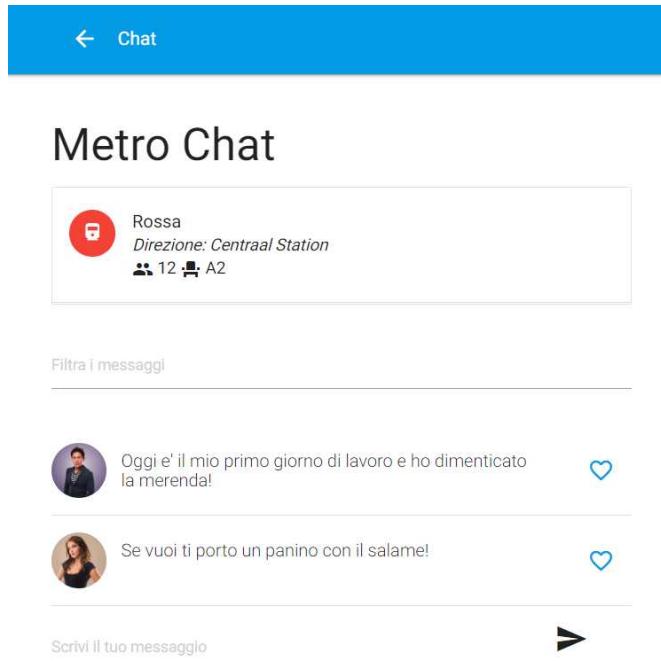


Figura 14.21 – Inserimento del campo di ricerca nel componente di dettaglio

Lo creiamo con le tecniche dei form reattivi, sfruttando il servizio `FormBuilder` giusto per fare pratica anche con questa modalità. Riportiamo solo le parti di interesse, visto che tutto il resto rimane identico a quello già sviluppato in precedenza.

```
treni/dettaglio/dettaglio.component.ts

//...
import { FormBuilder, FormGroup, Validators } from '@angular/forms';

@Component({
  selector: 'ca-dettaglio',
  template: `
    <div *ngIf="treno">
      <p> Chat Treno: {{treno.linea}}</p>
      <p> ID: {{treno.idt}}</p>
    </div>
    <form [formGroup]="searchForm">
      Cerca <input type="text"
        formControlName="search">
    </form>
    <ca-chat *ngFor="let chat of listachat" [msgIn]="chat"></ca-chat>
    <input type="text" #msg required>
    <button (click)="invioMsg(msg.value);msg.value=''">Invia</button>
  `})
  
```

```

export class DettaglioComponent implements OnInit {
  idtreno!: string;
  treno!: Metro;
  listachat!: Messaggio[];
  chatmsg!: Messaggio;
  errormsg!: any;
  searchForm!: FormGroup;

  constructor(private route: ActivatedRoute,
    private treniservice:TreniService,
    private chatservice:ChatService,
    private formBuilder: FormBuilder) {}

  ngOnInit() {
    this.idtreno = this.route.snapshot.paramMap.get('id')!;
    this.inizializzoForm();
    ...
  }

  inizializzoForm() {
    this.searchForm = this.formBuilder.group({
      search: ['']
    });
    this.searchForm.get('search')!.valueChanges.pipe(
      switchMap(str =>
        this.chatservice.getListachatObservable(this.idtreno,str));
    )
    .subscribe(risp => this.listachat = risp);
  }
  //...
}

```

Il cuore di tutto è all'interno della funzione `inizializzoForm()` dove abbiamo usato l'operatore `switchMap()` per intercettare due flussi di Observable: il primo legato all'inserimento dei dati dell'utente nel campo del form e il secondo rappresentativo della lista di messaggi.

Questi ultimi sono recuperati dal server tramite il metodo `getListaChatObservable()` del service `chat.service.ts`, che abbiamo modificato per poter passare, come secondo argomento, la stringa di ricerca di volta in volta inserita dall'utente.

Osserva inoltre l'uso di `this.searchForm.get('search').valueChanges` che produce un Observable con tutti i cambiamenti prodotti nel campo del form, in contrapposizione a `this.searchForm.get('search').value` che restituisce uno "snapshot" dei dati. Dovrai solo fare attenzione a inserire un punto esclamativo in quanto tra i possibili valori restituiti da `get()` c'è *null*.

```
service/chat.service.ts
// ...
getListaChatObservable(idt:string,
  txt:string = ''): Observable<Messaggio[]> {
  return this.http.get<Messaggio[]>(
    this.apiGetUrl+'?idt='+idt+'&idu=99&search='+txt)
    .pipe(
      map((risposta:any) => risposta['dati']),
      catchError(this.handleErrorObs)
    );
}
// ...
```

Chiaramente abbiamo dovuto effettuare il subscribe finale, che intercetta l'Observable interno e valorizza la variabile listachat, che permetterà all'utente di visualizzare la lista completa trovata.

Il vantaggio di switchMap() come dicevamo è che per ogni nuova lettera inserita, la precedente richiesta al server, non ancora evasa, è eliminata a favore di una nuova richiesta.

Difatti inserendo la stringa di ricerca “*dav*” solo l'ultima richiesta viene evasa come possiamo vedere nella console del browser.

Filter	<input type="checkbox"/> Hide data URLs	All	XHR	JS	CSS	Img	Mo
Name		Status	Type				
<input type="checkbox"/> ?idt=12345&idu=99&search=d		(canceled)	xhr				<u>zone-evergr</u>
<input type="checkbox"/> ?idt=12345&idu=99&search=da		(canceled)	xhr				<u>zone-evergr</u>
<input type="checkbox"/> ?idt=12345&idu=99&search=dav		200	xhr				<u>zone-evergr</u>

Figura 14.22 – Elenco richieste al server cancellate in seguito all'uso di switchMap()

Il risultato che vedremo sullo schermo sarà una lista ridotta di messaggi.

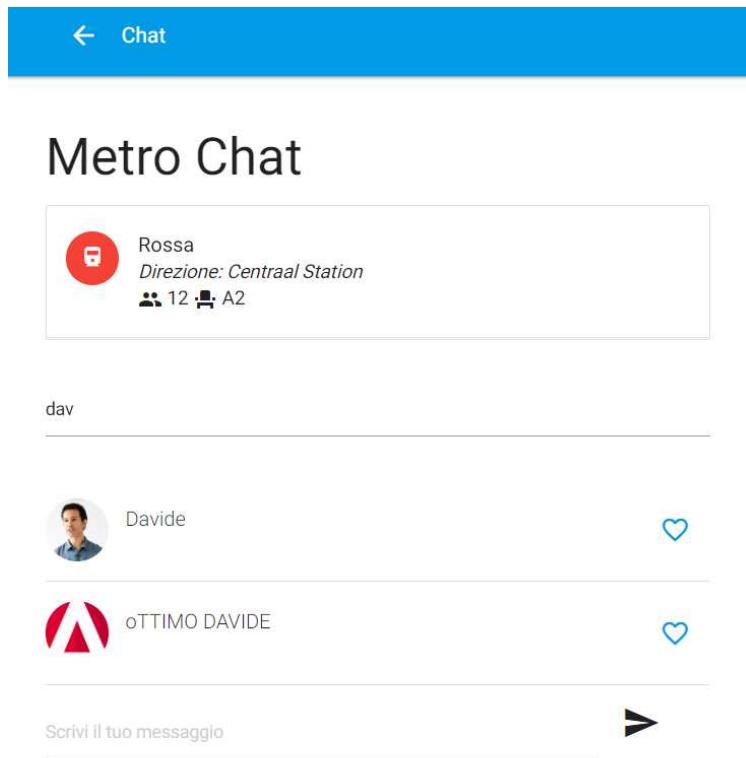


Figura 14.23 – Visualizzazione messaggi filtrati in base alla stringa di ricerca inserita dall’utente

Sicuramente un passo in avanti rispetto all’utilizzo di `flatMap()`, in cui tutte le richieste sono accodate e devono essere evase, ma ti renderai conto da solo che anche questa non è la soluzione ottimale .

L’ideale infatti è che si invii la richiesta solo dopo che l’utente ha inserito non meno di 2/3 caratteri, perché durante la digitazione spesso si cambia idea e si rischia di inviare molte richieste per ricercare messaggi che corrispondono a stringhe che non sono quelle che realmente l’utente vuole usare.

Per fare questo dobbiamo introdurre un altro operatore: `debounceTime()`.

Operatore debounceTime()

Le richieste consecutive ad un server dove sono memorizzati i dati da mostrare all’utente rappresentano uno dei principali motivi di rallentamento nell’esecuzione di un’applicazione e devono essere valutate attentamente per non creare delle esperienze utente insoddisfacenti.

Nei casi come quelli di flussi originati da eventi consecutivi, non tutti utili per lo scopo che si siamo prefissati, ci viene in aiuto un particolare operatore filtro: si tratta di `debounceTime()`.

Come dice lo stesso nome, ci permette di “ammortizzare” eventuali richieste multiple. Con il termine “ammortizzare” intendo l’eliminazione delle richieste emesse prima di un certo tempo minimo, usato come discriminante tra considerare o meno la richiesta.

Per ottimizzare il codice della nostra applicazione, dobbiamo quindi intervenire bloccando per un certo numero di millisecondi le richieste inviate in seguito alla pressione dei tasti della tastiera nel campo di input.

```
// ...
this.searchForm.get('search').valueChanges.pipe(
  debounceTime(300),
  switchMap(str =>
    this.chatService.getListChatObservable(this.idtreno, str);
  )
).subscribe(risp => this.listachat = risp);
// ...
```

Ho inserito un ritardo di 300 millisecondi che mi permette di attendere e recuperare più caratteri di ricerca. Dopo questo intervallo di tempo, parte l’effettiva richiesta al server, come puoi vedere dalla console del browser:

Filter		<input type="checkbox"/> Hide data URLs	All	XHR	JS	CSS	Img	M
Name	Status	Type	Initiator					
<input type="checkbox"/> ?idt=12345&idu=99&search=dav	200	xhr	<u>zone-evergi</u>					

Figura 14.24 – Invio della richiesta dopo l’attesa di 300 ms.

Se il server impiegasse più di 600 millisecondi nel rispondere, e l’utente nel frattempo avesse inserito altri caratteri, sarebbe eliminata la prima richiesta a favore della nuova che raccoglie tutti i caratteri fino a quel momento inseriti.

Un passo avanti sicuramente a livello di numero effettivo di richieste inviate al server. Fai attenzione a non inserire tempi troppo lunghi perché incideresti sul tempo di attesa.

14.8 Gestire gli errori

Fino ad ora in tutte le operazioni di subscribe, abbiamo sempre considerato solo il primo callback dell'Observer, quello che gestisce l'arrivo dei dati. Come dicevamo all'inizio, oltre ai dati l'Observer è in grado sia di gestire gli errori emessi dall'Observable sia di segnalare il termine dell'invio.

Il callback che passeremo come secondo parametro è quello che ci permette di intercettare e segnalare all'utente il messaggio di errore emesso dall'Observable.

```
dati$.subscribe(
  value => {},
  err => {} ,
  () => {}
);
```

Quando si sviluppa un'applicazione partendo da zero, si tende a non pensare agli effetti che può produrre un generico errore. Questo perché si è pienamente concentrati sulla logica. Questo però non significa che bisogna dimenticarsi di prevedere cosa fare nel caso qualcosa vada storto.

Ad esempio un tipico problema che si riscontra nelle applicazioni mobile, è la perdita del segnale wifi o la perdita della connessione dati. Fortunatamente accade raramente, ma potrebbe capitare proprio mentre stavamo attendendo la risposta del server in seguito all'invio di dati di registrazione di un modulo che ci ha occupati per due minuti.

In questi casi, l'iscrizione all'Observable termina in automatico con un errore e non c'è modo di effettuare nuovamente la richiesta precedente.

Questo obbliga l'utente ha compiere una nuova azione che potrebbe nel frattempo essere diversa dalla precedente se i dati del modulo si sono cancellati.

E' chiaro che dobbiamo segnalare all'utente l'errore in modo che possa prendere delle decisioni. Decisione che potrebbe essere o ripetere la richiesta o ahimè abbandonare definitivamente la nostra applicazione.

Ci viene in aiuto il metodo `catchError()` che intercetta eventuali errori di comunicazione tra la nostra applicazione e il server remoto dove sono memorizzati i dati da recuperare, prima che arrivino all'Observer.

Potrebbero essere errori legati a un malfunzionamento del server o legati all'assenza di connessione di rete.

Nell'esempio visto in precedenza, è il metodo `getListaChatObservable()` responsabile del recupero dei dati dal server, interrogando l'API messa a disposizione. Per intercettare gli errori nel Service abbiamo sfruttato `catchError()` e il metodo `handleErrorObs()` interno allo stesso Service, che restituisce l'errore all'Observable tramite `throwError()`.

In questo modo possiamo predisporre le righe per intercettarlo proprio all'interno del subscribe.

```
// ...
this.searchForm.get('search').valueChanges.pipe(
  debounceTime(300),
  switchMap(str =>
    this.chatservice.getListaChatObservable(this.idreno ,str);
  )
).subscribe(
  resp => this.listachat = resp,
  err => alert('Qualcosa è andato storto' + err)
);
// ...
```

Il callback passato come secondo parametro è quello che ci permette di intercettare l'errore e segnalare all'utente con un semplice `alert()`, il tipo di messaggio d'errore.

Come dicevamo all'inizio, l'errore provoca la cancellazione dell'Observable a cui ci eravamo iscritti, quindi impedisce fisicamente di recuperare i dati dal server remoto.

E se invece volessimo dare una seconda possibilità di richiesta senza far intervenire l'utente, come possiamo fare? Come sempre sfruttando un nuovo operatore della libreria RxJS: `retry()`

Quest'ultimo permette di rieseguire in automatico una o più iscrizioni all'Observable andato in errore, oltre le quali, se è ancora presente l'errore completare definitivamente la richiesta e segnala il problema all'utente. Il diagramma di Marble mostra chiaramente il flusso .

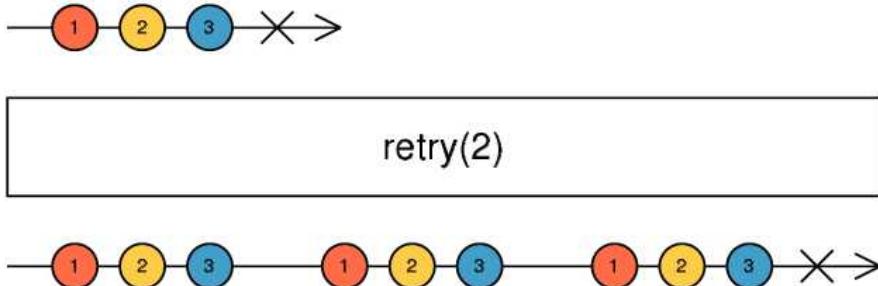


Figura 14.25 – Diagramma di Marble dell’operatore `retry()`

L’Osservabile originario ad un certo punto va in errore (X), ma essendoci l’operatore `retry()`, viene nuovamente eseguito un `subscribe` all’Osservabile sorgente. In assenza del `retry()`, il flusso terminerebbe subito al primo errore.

L’argomento richiesto dall’operatore, indica quante volte dobbiamo ripetere l’operazione di `subscribe`.

Applicando queste nozioni al metodo visto in precedenza, potremmo tentare di ripetere l’iscrizione per due tentativi, prima di terminare con l’errore.

```
//...
this.searchForm.get('search').valueChanges.pipe(
  debounceTime(300),
  switchMap(str =>
    this.chatService.getListChatObservable(this.idtreno, str).pipe(
      retry(2)
    )
  )
).subscribe(
  risp => this.listachat = risp,
  err => alert('Qualcosa è andato storto' + err)
);
// ...
```

In presenza di un errore, il `subscribe()` originario sarebbe perso, ma essendoci l’operatore `retry(2)` si riattiva e viene effettuata un nuovo `subscribe`.

Questo per altre 2 volte, quindi in totale avremo tre richieste. Se anche dopo il secondo tentativo, la richiesta non va a buon fine (es. la connessione non è ancora attiva), si passa alla visualizzazione dell’errore, gestito nel secondo parametro del metodo `subscribe()`, che lo segnalerà all’utente in qualche forma (es. messaggio toast).

Capitolo 15

Animare elementi di una vista

15.1 Introduzione al concetto di animazione

Il concetto di animazione applicato alle applicazioni web fa un po' sorridere rispetto a quello a cui siamo abituati quando vediamo un cartone animato. Chi di voi ricorda il grande Cavandoli e gli spot della pentola l'Agostina con la "linea". Se hai meno di 45 anni, prova a cercare su YouTube il carosello "la linea" e ti divertirai per qualche secondo.



Figura 15.1 - <http://www.youtube.com/watch?v=lhzSC8MzgGM>

Un tempo, nel web, il "principe" delle animazioni era il software Flash, da molti detestato per aver reso i siti una sorta di "circo" con effetti speciali. Tuttavia chi era in grado di usarlo con saggezza, dava origine a progetti di tutto rispetto. L'avvento dei dispositivi mobili, dell'HTML5 e le troppo falle di sicurezza, ne ha decretato la definitiva estinzione come plugin.

Ora, nella maggior parte dei siti che navighi, le animazioni si limitano allo spostamento di alcuni elementi della pagina, a effetti di transizione, rotazione, ingrandimento, luminosità. E' difficile ottenere effetti più complessi agendo solo sulle proprietà CSS3 di una pagina e senza usare software specifici.

Inoltre i giganti del web quali Google e Apple, hanno quasi imposto uno stile predefinito per le animazioni , sia sul mondo dei dispositivi mobili che nel mondo web, rendendo il tutto quasi standardizzato e per alcuni forse un po' noioso.

Detto questo, potremmo liquidare il capitolo con poche righe dicendo che, come spesso accade, a meno che tu o il tuo cliente non vogliate un'animazione iper personalizzata, non ha senso inventare l'acqua calda ed è sufficiente appoggiarsi ad una delle tante librerie nate sul web per semplificare la vita dei programmati.

Ad esempio, nel mondo dei classici siti web, sicuramente hai sentito parlare di *animate.css*, uno dei tanti progetti che permette di applicare , con pochi click, più di 30 diverse animazioni agli elementi della pagina.

Anche nel mondo Angular, con la stessa facilità si possono applicare gli stessi effetti, sfruttando l'ottimo lavoro di Chris Filipowski che ha creato una libreria di animazioni che prendono spunto proprio da *animate.css*

Usando poche righe di codice, già puoi vedere animati i vari elementi di una pagina. La libreria è molto semplice da usare e la puoi trovare a questo link:

<https://github.com/filipows/angular-animations>

In questo capitolo, tuttavia, inizieremo ad esplorare i passaggi fondamentali per creare da zero le proprie animazioni da applicare ai diversi elementi di una vista, e alla transizione tra pagine di un'intera applicazione.

15.2 Installare la libreria per le animazioni

Tutte le nuove applicazioni che si creano con Angular CLI, presentano nella cartella *@angular* di *node_module*, la libreria necessaria a inserire delle animazioni.

Lo puoi verificare anche aprendo il file *package.json* e controllando che tra le dipendenze esista la libreria *animations*:

```
packages.json
"dependencies": {
  "@angular/animations": "~X.X.X",
  ...
}
```

Nel caso non fosse presente, dovrai lanciare la classica riga per l'importazione:

```
npm install @angular/animations@latest -save
```

Verificato questo, vediamo gli ulteriori passaggi che dovremo seguire per dotare una qualsiasi applicazione di effetti animati.

Le prime modifiche dovranno essere apportate al file `app.module.ts`, in modo da importare il modulo `BrowserAnimationsModule` dal percorso `@angular-platform/animations`. Poi dovremo aggiungerlo all'interno della lista dei moduli tramite la proprietà `import` del decoratore `@NgModule()`.

`app.module.ts`

```
import { BrowserAnimationsModule } from '@angular/platform-browser/animations';

@NgModule({
  declarations: [
    ...
  ],
  imports: [
    BrowserModule,
    BrowserAnimationsModule,
    ...
  ],
  providers: [TreniService, ChatService],
  bootstrap: [AppComponent]
})

export class AppModule { }
```

Questa è l'unica modifica da effettuare a livello generale per l'intera applicazione.

Ora possiamo concentrarci sui singoli componenti e sulla definizione e creazione delle animazioni che vogliamo introdurre nell'applicazione. Partiremo con analizzare i singoli attori in gioco, in modo da avere una panoramica degli strumenti che Angular ci mette a disposizione. Poi vedremo come combinarli insieme al fine di ottenere i risultati voluti.

Sfrutteremo il palcoscenico dell'applicazione che abbiamo creato fino ad ora in modo da vedere in anteprima gli effetti delle nozioni che spiegheremo.

15.3 Definire i nomi delle applicazioni

Il primo passo è decidere quale pezzo dell'applicazione MetroChat dovrà avere degli elementi da animare. Possono essere elementi interni a un componente, oppure l'intero componente nel passaggio da una vista all'altra. Queste sono le due tipiche situazioni che dovrai affrontare, quindi sono quelle che spiegheremo nel corso delle prossime pagine.

Senza entrare nel dettaglio del tipo di effetto che vogliamo ottenere, cerchiamo di individuare il componente e gli elementi della sua vista che intendiamo animare.

Sapendo che la prima pagina visualizzata è quella associata al componente *treni.component.ts*, dovremo aggiungere una nuova proprietà al decoratore `@Component`.

La proprietà è `animations` che valorizzeremo con un array costituito dalla definizione delle animazioni che intendiamo applicare ai diversi elementi della vista.

```
treni/treni.component.ts

//...
@Component({
  selector: 'ca-treni',
  templateUrl: './treni.component.html',
  animations: [
    definizione_animazioneA,
    definizione_animazioneB,
    ...
  ]
})
// ...
```

Ogni animazione sarà caratterizzata da funzioni `trigger()` che impareremo a progettare nel corso del capitolo e che, come suggerisce il nome, permettono di identificare in modo univoco le diverse animazioni e di agganciarle a elementi del template.

Noi, per ora, non creeremo dei file esterni riutilizzabili da altri componenti, ma definiremo il corpo delle animazioni direttamente nel decoratore.

Come sempre, per poter utilizzare tutte le funzioni dedicate alla creazione di animazioni, dovremo importarle nel componente. Si trovano tutte all'interno della libreria `@angular/animations`.

treni.component.ts

```
import { trigger, state, style, transition, animate }
        from '@angular/animations';
// ...
```

Abbiamo incluso anche quelle che ci serviranno da qui in avanti. Qui sotto possiamo vedere l'elenco completo. Gran parte di queste ti diventeranno familiari i non appena avrai finito di leggere il capitolo.

Funzione	Descrizione
trigger()	E' il metodo che permette di avviare l'animazione e serve come "contenitore" per tutte le altre chiamate alle funzioni di animazione. Permette di specificare il nome dell'animazione che sarà usato nel template del componente.
style()	Permette di definire uno o più stili CSS da usare nell'animazione. E' equivalente all'impostazione dello stile di un elemento della pagina, tramite un foglio CSS.
state()	Permette di definire uno stato dell'animazione, caratterizzato da un certo numero di stili CSS da applicare durante il passaggio dell'animazione da uno stato all'altro.
keyframe()	Permette di definire una sequenza di cambi di stile, legati alla funzione animate().
animate()	Permette di specificare l'andamento dell'animazione, in termini di durata, ritardo, effetto, e stile.
transition()	Definisce la sequenza di animazioni tra due stati.
group()	Definisce un gruppo di animazioni da eseguire in parallelo. Solo al termine di tutte quelle interne, l'animazione complessiva può continuare o terminare.
query()	Permette di identificare un elemento del DOM dell'attuale vista, a cui applicare uno stile o animazione.
stagger()	Scaglionata l'ora d'inizio dell'animazione per più elementi.

sequence()	Specifica una lista d'animazioni da eseguire in successione, una alla volta.
animation()	Produce un'animazione riutilizzabile che può essere invocata da <code>useAnimation()</code> .
useAnimate()	Attiva un'animazione riutilizzabile.
animateChild()	Permette l'esecuzione di animazioni su componenti figlio, che saranno eseguite con lo stesso lasso di tempo del componente padre.

Il passo successivo è quello di decidere cosa animare e come. Partiamo con la schermata iniziale che l'utente vede non appena accede all'applicazione.

MetroChat

In arrivo Preferiti Login

Treni in Arrivo a questa stazione

Qui sotto puoi vedere l'elenco dei treni in arrivo presso questa ipotetica stazione. Cliccando sul simbolo del treno, potrai accedere al dettaglio, in cui vedrai la lista degli utenti che stanno chattando. Potrai inserire il tuo messaggio e aggiungere quello degli altri alla lista dei preferiti, in modo da ricontrattarli privatamente.

Gialla
Direzione: Zuid
• 4 C4 00:29

Verde
Direzione: Isolatorweg
• 3 C4 01:17

Rossa
Direzione: Centraal Station
• 12 A2 00:29

A

B

Treni appena partiti

Qui sotto puoi vedere la lista dei treni che sono appena partiti e che hai perso.

Figura 15.2 – Individuazione delle sezioni da animare all'apertura dell'applicazione

L'idea che abbiamo in mente è di animare la scritta individuata da A nella *figura 15.2 - Treni in arrivo...* - con un effetto di dissolvenza in ingresso, facendola partire da fuori dello schermo sul bordo destro, con un'opacità pari a "0.3" (quasi invisibile all'occhio), per arrivare, a fine animazione, nella posizione attuale di *figura 15.2*, con un'opacità pari a "1".

Analogo discorso per il blocco individuato da B, che rappresenta la lista dei treni, ma questa volta creando solo un effetto di dissolvenza in ingresso, partendo da un'opacità pari a "0.3" nella fase di caricamento dei dati, e portandola a "1" dopo che i dati sono stati completamente ricevuti. Qui sotto uno schema dello stato iniziale e finale dell'animazione che intendiamo realizzare.

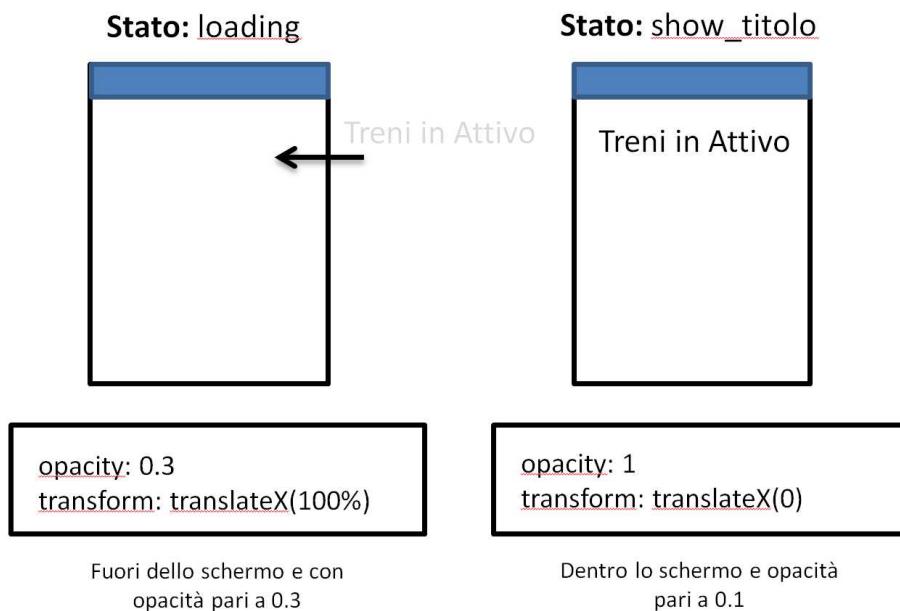


Figura 15.3 – I due stati corrispondenti all'animazione del titolo dell'applicazione

Ora dobbiamo decidere i nomi da dare alle due animazioni. Per la nostra applicazione potremmo scegliere per la prima "titolo_in" e per la seconda "listatreni_in".

```
//...
@Component({
  selector: 'ca-treni',
  templateUrl: './treni.component.html',
```

(continua)

treni/treni.component.ts

```

  animations: [
    trigger('titolo_in', [ ... ]),
    trigger('listatreni_in', [ ... ])
    ...
  ]
})
// ...

```

Abbiamo usato la prima funzione utile della libreria `@angular/animations`, ossia `trigger()` che è quella che ci permette di dare un nome alle diverse animazioni che intendiamo creare. Oltre a questo, ci permette di definire il comportamento caratteristico dell'animazione, in particolare gli stati di cui è costituita e le transizioni tra uno stato e l'altro.

La sintassi base è questa:

```
trigger('nomeanimazione', [caratteristiche_animazione])
```

Il nome ci servirà per agganciare l'animazione, o ai vari elementi del componente o all'intero componente.

Le caratteristiche dell'animazione sono rappresentate dagli stati e dalle transizioni tra questi. In sostanza si tratta dell'insieme delle proprietà CSS e il relativo andamento nel tempo.

15.4 Gli stati di un'animazione

Come dicevamo, il cuore di un'animazione è rappresentato dagli stati, che possono essere pensati come ai fotogrammi di una pellicola. Nei software usati per lo sviluppo di animazioni vengono chiamati “keyframe” o fotogrammi chiave.

Ad esempio il progetto di un cartone animato, un tempo prevedeva diversi team di animatori. Semplificando, la prima squadra era quella responsabile della creazione dei fotogrammi chiave, equivalenti - a grandi linee - alle scenette di un classico fumetto.

La seconda squadra era quella responsabile della creazione di tutti i fotogrammi per passare da una scenetta all'altra (transizione) e quindi creare l'effetto del movimento.

Quest'ultimo si origina proprio perché le informazioni presenti tra due stati consecutivi sono diverse tra loro. Ad esempio, potrebbe cambiare la posizione di un occhio o di un piede o di una scritta.

Nel caso dei film d'azione, un fotogramma potrebbe visualizzare il tuo attore preferito mentre sta saltando da bordo di un palazzo, il fotogramma successivo mentre inizia ad essere sospeso nel vuoto con un piede, poi con l'altro e così via.

In Angular, si definisce uno *stato* impostando un nome univoco e definendo le proprietà CSS finali che dovrà avere l'elemento a cui si applicherà. Tali proprietà saranno mantenute anche al termine dell'animazione che coinvolge l'elemento.

La funzione che Angular ci mette a disposizione è `state()` e richiede tipicamente due argomenti: il primo è la stringa che rappresenta il nome da dare allo stato per referenziarlo nel codice e il secondo è l'insieme delle proprietà CSS che caratterizzeranno quello stato al termine dell'animazione in cui è inserito.

```
state('nome_dello_stato', stile_proprietà_CSS)
```

Nel nostro caso, le due animazioni potrebbero entrambe avere due stati che identificano rispettivamente la fase d'apertura dell'applicazione, in cui si dà inizio all'animazione, e la fase in cui l'animazione si conclude, corrispondente al recupero di tutti i dati dall'API.

Per l'animazione del titolo, potremmo usare i nomi “loading” e “show_titolo”, mentre per la lista dei treni rispettivamente “loading” e “show_treni”.

Questi diversi stati, chiaramente, dovranno essere definiti all'interno della definizione dell'animazione che riguarda il titolo e la lista dei treni.

```
//...
animations : [
  trigger('titolo_in', [
    state('loading', STILE_PROP_CSS),
    state('show_titolo', STILE_PROP_CSS),
  ]),
  trigger('listatreni_in', [
    state('loading', STILE_PROP_CSS),
    state('show_treni', STILE_PROP_CSS),
  ])
]
//...
```

E' chiaro che, a differenza dei film in cui è l'attore che compie il movimento, nel caso delle applicazioni, saremo noi a decidere ogni posizione, spostamento, e cc. che dovranno avere i diversi elementi.

Il cambiamento dello stato potrà avvenire in diversi modi: in seguito al verificarsi di una particolare condizione (es. al termine del caricamento dei dati da una sorgente remota), in seguito a un'azione fatta dall'utente, o nel passaggio da una pagina all'altra.

15.5 Definire le proprietà CSS di uno stato o animazione

Le classiche proprietà che si tendono a usare per animare gli elementi di una vista, sono lo spostamento orizzontale, verticale, l'opacità, l'ingrandimento, i colori. Combinando insieme queste diverse grandezze, possiamo ottenere diversi stati.

Tutto questo è possibile grazie a una funzione molto versatile, che può essere usata singolarmente o all'interno di altre funzioni. Si tratta di `style()`, che nella forma base, richiede tante coppie chiave/valore rappresentative delle proprietà CSS che deve avere un elemento in quello specifico stato o durante un'animazione.

La sintassi è:

```
style({ proprietà_da_animare })
```

Alcuni esempi possono essere:

```
// imposto un valore stringa per le proprietà CSS
style({ background: "red", color: "blue" })

// imposto un valore numerico
style({ width: 100, height: 0 })
```

NB: Le proprietà CSS costituite da due o più parole, devono essere scritte eliminando il trattino e usando la notazione CamelCase (es.`backgroundColor` invece di `background-color`)

E' possibile usare anche il simbolo dell'asterisco, per indicare il valore di una proprietà che viene ereditata dallo stile originale dell'elemento.

Per il titolo della pagina nello stato “loading”, sfrutteremo le proprietà legate all’opacità e alla traslazione sull’asse X:

```
state('loading', style({
    opacity: 0,
    transform: 'translateX(100%)'
})
)
```

In questo modo, la scritta si troverà al di fuori dello schermo, e sarà invisibile. E’ chiaro che questo sarà lo stato iniziale non appena avvieremo l’applicazione. In assenza di un trigger che cambi questo stato, la scritta rimarrà invisibile all’utente.

Analogo discorso per lo stato finale. L’opacità sarà portata al valore “1” al fine di renderla visibile. Non abbiamo setta to il valore della proprietà `transform`, perché in questo modo Angular userà lo stesso posizionamento che assume l’elemento nel DOM, nel suo stato predefinito.

```
state('show_titolo', style({opacity: 1}))
```

Mettendo insieme i diversi pezzi, otteniamo la definizione base della nostra prima animazione da applicare al titolo della pagina:

```
//...
animations : [
  trigger('titolo_in', [
    state('loading', style({
      opacity: 0,
      transform: 'translateX(100%)'
    })
  ),
  state('show_titolo', style({opacity: 1}))
])
]
//...
```

Bisogna sottolineare il fatto che questa animazione non contiene ancora degli stati intermedi, senza i quali l’utente vedrà inevitabilmente un passaggio netto da uno stato all’altro. Avremo modo di perfezionarla quando introdurremo il concetto di transizioni.

Le stesse identiche considerazioni varranno anche per il blocco che mostra la lista dei treni. In questo caso agiremo solo sull'opacità, quindi mettendo insieme le due animazioni, otterremo:

```
treni/treni.component.ts

//...
import { trigger, state, style, transition, animate }
        from '@angular/animations';
@Component({
  selector: 'ca-treni',
  templateUrl: './treni.component.html',
  animations : [
    trigger('titolo_in', [
      state('loading', style({
        opacity: 0,
        transform: 'translateX(100%)'
      })
    ),
    state('show_titolo', style({opacity: 1}))
  ]),
    trigger('listatreni_in', [
      state('loading', style({opacity: 0})),
      state('show_treni', style({opacity: 1})),
    ])
  ]
})
//...
```

Il passo finale per vedere se il tutto funziona, è quello di capire come applicarla a gli elementi presenti nel template del componente, iniziando dal titolo, per poi passare anche al blocco che visualizza i treni.

15.6 Collegare l'animazione all'elemento nella vista

Una volta definita l'animazione e assegnato un nome grazie alla funzione `trigger()`, possiamo spostarci nella vista del componente e individuare l'elemento che vogliamo animare.

A questo applicheremo l'animazione e lo stato iniziale, sfruttando questa notazione:

```
<TAG_elemento [@NOME_TRIGGER_SCELTO] = 'VAR_STATO_ANIMAZIONE'>
</TAG_elemento>
```

Nel nostro caso dovremo modificare il tag `<h1>` dove è inserito il titolo “Treni in Arrivo”.

```
treni.component.html

//...
<h1 [@titolo_in]="stato_titolo">Treni in arrivo ...</h1>
//...
```

All’interno delle parentesi quadre, abbiamo inserito il nome del trigger `titolo_in` progettato in precedenza, mentre a destra dell’uguale dovremo aggiungere il nome di una variabile definita nella classe del componente e che conterrà la stringa rappresentativa dei nomi assegnati ai vari stati dell’animazione.

Potremmo chiamarla `stato_titolo`. E’ proprio grazie a questa che riusciremo a commutare da uno stato all’altro e quindi ad animare la scritta.

Nella fase iniziale, non appena il componente è creato, l’animazione della scritta dovrà essere impostata nello stato “loading” (posizionamento fuori dallo schermo), mentre al termine del caricamento dei dati, dovremo modificarla nello stato “show_titolo”.

E’ chiaro quindi che dovremo predisporre nella classe, un’istruzione per inizializzare la variabile e una per modificarla. In quale punto? Sicuramente al termine del recupero dei dati dal server remoto, quindi all’interno del metodo `getListaMetroObservable()`:

```
treni/treni.component.ts

//...
export class TreniComponent implements OnInit {
  stato_titolo: string = 'loading';
  stato_lista: string = 'loading';
  ...
  getListMetroObservable() {
    this.treniservice.getListMetroObservable()
      .subscribe(
        p => {
          this.listametro = p;
          this.stato_titolo = 'show_titolo';
          this.stato_lista = 'show_lista';
        },
        error => this.errormsg = error
      );
  }
//...
```

Abbiamo inizializzato le due variabili con i nomi delle stringhe rappresentative degli stati iniziali.

Tutto il codice realizzato fino ad ora sarà:

```
treni/treni.component.ts

//...
import { trigger, state, transition, style, animate }
        from '@angular/animations';

@Component({
  selector: 'ca-treni',
  templateUrl: './treni.component.html',
  animations : [
    trigger('titolo_in', [
      state('loading', style({
        opacity: 0,
        transform: 'translateX(100%)'
      }),
      state('show_titolo', style({opacity: 1}))
    ]),
    trigger('listatreni_in', [
      state('loading', style({opacity: 0})),
      state('show_lista', style({opacity: 1}))
    ])
  ]
})
export class TreniComponent implements OnInit {
  stato_titolo: string = 'loading';
  stato_lista: string = 'loading';
  listametro: Metro[];
  trenipartiti: string;
  errormsg!: any;

  constructor(private router: Router, private treniservice: TreniService) {
    this.trenipartiti = '';
    this.listametro = [];
    this.now = new Date().getTime();
  }

  ngOnInit() {
    this.getListaMetroObservable();
  }

  getListaMetroObservable() {
    this.treniservice.getListaMetroObservable()
      .subscribe(
        p => {
          this.listametro = p;
          this.stato_titolo = 'show_titolo';
          this.stato_lista = 'show_lista';
        },
        error => this.errormsg = error      (continua)
      )
  }
}
```

```

        );
    }
    setMetro(id: string) {
        this.router.navigate(['inarrivo/dettaglio', id]);
    }
    partiti(id: string) {
        this.trenipartiti += "|" + id;
    }
}

```

mentre il template sarà:

treni/treni.component.html

```

<ca-menu></ca-menu>
<h1 [@titolo_in]="stato_titolo">Treni in Arrivo</h1>
<div class="listtreni" [@listatreni_in]="stato_lista">
    <ca-metro *ngFor="let metro of listametro"
        (inpartenza)="partiti($event)"
        [datiIn]="metro"
        [ora]="now"
        (click)="setMetro(metro.idt)">
    </ca-metro>
    <div *ngIf="errormsg">{{errormsg}}</div>
</div>
<p>{{trenipartiti}}</p>

```

15.7 Il passaggio da uno stato all'altro

Come sottolineato in precedenza, l'animazione così creata presenta un grosso problema: non sembra un'animazione. Quello che manca infatti è la simulazione del movimento, che ci permette di passare da uno stato all'altro in modo fluido e non repentino.

Sarebbe alquanto dispendioso in termini di tempo creare decine e decine di stati intermedi che indichino la posizione della scritta nello spostamento da destra verso sinistra dello schermo. Analogi discorsi per simulare l'andamento dell'opacità da valori prossimi a “0” al valore “1”.

Fortunatamente gli sviluppatori di Angular hanno pensato anche a questo e hanno creato la funzione `transition()`, che permette di stabilire in che modo debba avvenire il passaggio da uno stato all'altro di un'animazione.

Nella forma base, richiede almeno due argomenti: il primo è quello che ci permette di definire quali stati sono coinvolti nella transizione, mentre il secondo può essere o

un array con i diversi step dell'animazione o un singolo step (es. durata, tipo di andamento).

```
transition("stato_partenza => stato_arrivo", [step_animazione])
```

Nel nostro caso, dovremo stabilire come far avvenire rispettivamente il passaggio dallo stato “loading” allo stato “show_titolo” dell’animazione “titolo_in” e il passaggio dallo stato “loading” allo stato “show_treni” dell’animazione “listatreni_in”.

Per il primo argomento, la notazione da usare è molto intuitiva, nel senso che si inserisce una stringa con i nomi degli stati coinvolti , separati da una freccia che indica in che direzione va applicata la transizione.

Transizione	Descrizione
statoA => statoB	Transizione dallo stato A allo stato B
statoA <=> statoB	Transizione dallo stato A allo stato B e viceversa

NB: Il simbolo di uguale (=) unito alla freccia a sinistra o a destra (< >) - chiamato in gergo fat arrow - deve essere inserito senza spazi.

E’ possibile anche applicare dei valori predefiniti come void e * che hanno significati particolari:

Transizione	Descrizione
* => *	Transizione da un qualsiasi stato ad un altro
void => *	Transizione da un uno stato di un elemento non ancora presente nella pagina, a un o stato qualsiasi .
:enter	Transizione in ingresso (es. quando aggiungo un nuovo elemento) equivalente a: void => *
:leave	Transizione in uscita (es. quando elimino un elemento) equivalente a: * => void

Per quanto riguarda il secondo argomento, nella maggior parte dei casi si indica un singolo step inserendo la funzione predefinita `animate()`:

```
transition("stato_partenza => stato_arrivo",
           animate('durata ritardo andamento', stile))
```

Quest'ultima richiede due argomenti che definiscono rispettivamente alcuni dati specifici dell'animazione come la durata, il ritardo, l'effetto e come secondo argomento opzionale, lo stile CSS finale che dovrà avere l'elemento a cui si applica l'animazione.

```
animate('durata ritardo effetto', stile)
```

La durata e il ritardo possono essere espresse con un numero (millisecondi) oppure con una stringa indicante anche l'unità di misura (es. "1s", "5ms"). L'effetto invece è l'andamento temporale dell'animazione, rappresentato da "curve di bezier", e può assumere dei valori predefiniti come:

- `ease-in` - lento all'inizio e in accelerazione verso la fine dell'animazione
- `ease-out` - veloce all'inizio e in decelerazione verso la fine dell'animazione
- `cubic-bezier(0,.5,1,.5)` - andamento veloce all'inizio e alla fine. Rallentato a metà animazione

Per sperimentare gli effetti dei diversi andamenti, puoi collegarti a questo link:
<https://cubic-bezier.com>

Nella maggior parte dei casi userai il primo argomento (durata ritardo andamento) con almeno indicata la durata dell'animazione. Tuttavia è possibile definire anche uno specifico stile, come avremo modo di vedere quando parleremo di come applicare le animazioni al cambio pagina.

Giusto per vedere qualche ulteriore esempio:

- `animate(500)` - Imposto solo la durata dell'animazione lineare pari a 500 millisecondi
- `animate("1s")` - Durata animazione espressa come stringa in secondi
- `animate("100ms 0.5s")` - Durata pari a 100 ms, con un ritardo alla partenza di 500 ms
- `animate("5s ease-in")` - Durata 5 secondi, andamento "easing in"
- `animate("5s 10ms cubic-bezier(.17,.67,.88,.1)")` - Durata di 500 ms con ritardo di 10 ms, e andamento stabilito dalla curva di bezier

In merito al secondo argomento rappresentativo dello stile, generalmente si usa quando non conosciamo a priori lo stato finale e di conseguenza non sappiamo che stile applicare.

Spesso trova utilizzo nel passaggio da `* => void` ossia `:leave` o quando si vogliono creare animazioni più complesse con `query()`, `group()` come avremo modo di vedere successivamente.

Ad esempio, se volessimo animare un elemento della vista non appena questo viene rimosso dal DOM, potremmo sfruttare `:leave` oppure `* => void`, e decidere lo stile da associare a fine animazione , senza la necessità di indicare un preciso stato finale .

Infatti se abbiamo deciso di farlo uscire dallo schermo verso destra, con un'animazione lineare di durata 500 ms, potremmo scrivere:

```
trigger('elemento_out', [
  transition('* => void', [
    animate(500, style({ transform: 'translateX(100%)' }))
  ])
])
```

Viceversa, se volessimo animare un componente non ancora presente nel DOM, quindi con uno stato iniziale sconosciuto, in modo che appaia da sinistra dello schermo con un'opacità pari a “0” e raggiunga la posizione finale dopo 500ms , individuata da uno stato chiamato “in”, potremmo scrivere:

```
trigger('elemento_in', [
  state('in', style({ opacity: 1, transform: 'translateX(0)' })),
  transition('void => in', [
    style({ opacity: 0, transform: 'translateX(-100%)' }),
    animate(500)
  ])
])
```

Chiariti questi concetti, cerchiamo di collegare tutte le nozioni viste fino ad ora al fine di applicare una transizione tra i due stati creati in precedenza .

All'elemento della pagina che contiene il titolo “Treni in arrivo” potremmo applicare una transizione dallo stato “loading” allo stato “show_t itolo” di durata pari a 300ms, con un andamento veloce all'inizio e più lento verso la fine (ease-in).

```
transition("loading => show_titolo", animate("300ms ease-in"))
```

Per la lista treni invece , impostiamo una durata un po' maggiore pari a 2 sec.:

```
transition("loading => show_treni", animate("2s ease-in"))
```

Il codice che include anche le transizioni, diventerà:

```
// ...
animations : [
  trigger('titolo_in', [
    state('loading', style({
      opacity: 0,
      transform: 'translateX(100%)'
    }),
    state('show_titolo', style({opacity: 1})),
    transition('loading => show_titolo',
              animate('300ms ease-out'))
  ),
  trigger('listatreni_in', [
    state('loading', style({opacity: 0})),
    state('show_lista', style({opacity: 1})),
    transition('loading => show_lista', animate('2s ease-in'))
  ])
]
// ...
```

L'effetto risultante sarà, un titolo che compare con uno spostamento da destra verso sinistra, e una lista dei treni che appare con una dissolvenza in ingr esso.

Questo chiaramente non appena la lista dei treni è recuperata per intero , quindi non appena avviene il cambio di stato, ossia sono eseguite le due istruzioni:

```
this.stato_titolo = 'show_titolo';
this.stato_lista = 'show_lista';
```

interne alla funzione `getListaMetroObservable()`.

Riassumendo tutto quello visto fino ad ora , possiamo dire che siamo riusciti a definire e applicare due distinte animazioni, combinando insieme le funzioni `trigger()`, `state()`, `transition()` e `animation()`.

Questi concetti possono essere applicati , non solo a qualsiasi elemento della pagina ma anche in seguito ad un evento di cambio stato non necessariamente pilotato da noi. Tipico esempio è quello legato all'inserimento di nuovi elementi nel DOM, per cui abbiamo a disposizione gli stati :enter e :leave.

Capitolo 16

Animare le transizioni tra pagine

16.1 Cosa si intende per transizione tra pagine

Altra tipica situazione che dovremo affrontare, è l'animazione legata a un cambio pagina. Nella nostra applicazione abbiamo poche pagine, quindi risulterà più semplice capire a fondo il meccanismo di funzionamento.

Rispetto alle animazioni di singoli elementi della pagina, subentrano tuttavia delle complicazioni che devono essere affrontate singolarmente. Prima di addentrarci nella scrittura del codice, cerchiamo di capire bene gli attori in gioco e l'effetto dell'animazione che vogliamo ottenere.

L'idea di movimento che abbiamo in mente – figura 16.1 – è che non appena l'utente clicca su un treno per visualizzare la lista degli attuali messaggi chat, la lista dei treni – TreniComponent – si animi spostandosi interamente verso sinistra, facendo posto all'ingresso da destra, della pagina di dettaglio – DettaglioComponent.

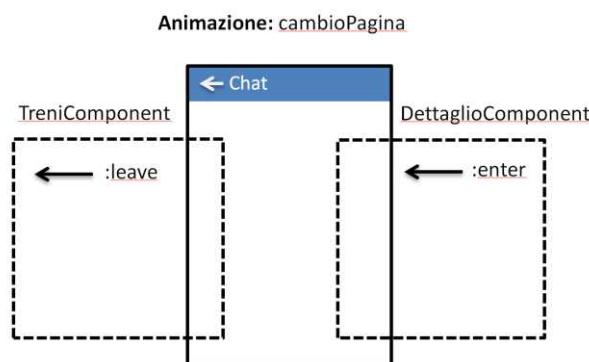


Figura 16.1 – Dettaglio animazione “cambioPagina”

E' chiaro quindi che in un certo istante, seppur breve, le due pagine coesisteranno sullo schermo e saranno entrambe visibili all'utente. Il problema è che a livello di CSS le due pagine occupano sempre l'intera finestra e si posizionano all'interno del contenitore `<router-outlet>`.

La domanda che sorge spontanea è come possiamo visualizzarle insieme se nel momento in cui cambio route, il componente viene distrutto?

Ammesso di risolvere questo problema, l'altro problema che sorge è come animarle in modo che una si sposti verso sinistra e l'altra prenda il posto della pagina uscente , entrando da destra. Se ipotizziamo che nel DOM le due pagine coesistano e siano come due blocchi con posizionamento assoluto, con poche proprietà CSS riusciremo a ottenere l'effetto desiderato.

La proprietà CSS da usare , infatti, è `left`, che come ben saprai, applicata a un elemento con posizionamento assoluto, permette di spostarlo al di fuori del layout dello schermo, rendendolo quindi invisibile all'utente.

Nel nostro caso dovremo partire con un elemento visibile, ossia il componente che dovrà uscire – `TreniComponent` - e uno invisibile, ossia il componente che dovrà entrare – `DettaglioComp`.

Giocando con questa proprietà e con gli stati iniziali e finali, otterremo l'effetto di spostamento voluto.

Si deve quindi escogitare un “trucco” CSS per far apparire insieme le due pagine. Il trucco è molto semplice , perché sarà sufficiente applicare un'animazione al contenitore delle pagine visualizzate all'interno di `<router-outlet>`, in modo da applicare, in contemporanea, un posizionamento assoluto a entrambe le pagine oggetto dell'animazione.

16.2 Configurare le route

Al fine d'indicare ad Angular quale stato applicare alle due pagine da animare, dobbiamo sfruttare un nuovo parametro di configurazione delle route.

Questo perché tutte le pagine sono visualizzate all'interno di un contenitore `<router-outlet>` quindi dovremo far scattare il trigger nel contenitore padre e passare a questo di volta in volta un nuovo stato.

L'unica possibilità che abbiamo è sfruttare le route e prevedere sia una variabile di classe che si aggiorni al nuovo stato , non appena cambiamo route , sia una variabile nel template in grado di intercettare le informazioni passate nella route.

In merito a quest'ultima, la proprietà da sfruttare è `data`, che valorizzeremo con un oggetto che rappresenta il nome dello stato che vogliamo applicare a 1 componente abbinato alla route .

Ad esempio , per la prima route presente nel file `app.module.ts`, associata al path „inarrivo”, potremmo scrivere:

```
//...
{
  path: 'inarrivo',
  component: TreniComponent,
  data: { statoanim: 'start' }
}
//...
```

`app.module.ts`

I nomi delle proprietà che inseriremo all'interno di `data`, sono del tutto arbitrarie e possono essere recuperate nella classe sfruttando la direttiva `RouterOutlet` e in particolare la proprietà `activatedRouteData`.

Per fare questo, nel template del componente `app.component.ts`, dove abbiamo inserito il contenitore con selettore `<router-outlet>`, aggiungeremo a quest'ultimo l'identificatore `outlet` e lo associeremo ad una variabile del template che ci permetterà di passarla ad altre funzioni, ad esempio a quella predisposta per far partire l'animazione in seguito al cambio dello stato.

```
<router-outlet #out = "outlet"></router-outlet>
```

`app.component.html`

Nel nostro caso abbiamo usato una variabile nel template di nome `out` valorizzata con l'identificatore predefinito `outlet`. Il passo successivo sarà quello di creare un contenitore delle pagine che definisca le animazioni da usare , sfruttando la valorizzazione di una variabile di stato.

16.3 Impostare il trigger a livello di contenitore pagina

Ora che sappiamo recuperare i diversi stati tramite le route, possiamo applicare gli stessi concetti visti in precedenza a livello di animazione di un singolo

componente. Nel nostro caso, dobbiamo animare l'intera pagina, quella presente all'interno del contenitore `<router-outlet>`.

Non possiamo applicare il trigger direttamente a quest'ultimo, ma dovremo creare un contenitore padre. Ci spostiamo quindi all'interno del template del componente `app.component.ts` e scriveremo:

```
app.component.html  
  
<main [@cambioPagina]="getStatoAnimazione(out)">  
  <router-outlet #out = "outlet"></router-outlet>  
</main>
```

L'animazione l'abbiamo chiamata “cambioPagina” ed è stata applicata ad un tag HTML5 `<main>` che conterrà proprio l'intera sezione dell'applicazione che vogliamo animare.

Questo significa che anche il menu dell'applicazione sarà animato. Vedremo in seguito come toglierlo dalle singole pagine.

Il risultato della chiamata al metodo `getStatoAnimazione()`, sarà quello che ci permetterà di impostare di volta in volta lo stato dell'animazione, a seconda dei dati recuperati dalla route.

Questi ultimi potranno essere estratti, passando al metodo proprio la variabile `out` di nome `out` definita nel template. Quest'ultima infatti sarà di tipo `RouterOutlet`, e questa volta tramite la proprietà `activatedRouterData`, riusciremo ad accedere ai dati statici della route.

```
app.component.ts  
  
import { Component, OnInit } from '@angular/core';
import { RouterOutlet } from '@angular/router';
import { trigger, state, style, animate, transition }
  from '@angular/animations';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  animations: [
    ...
  ]
})
export class AppComponent implements OnInit {
  title: string;
  constructor() {
    this.title = 'MetroChat con dettaglio';
  }
}
```

(continua)

```

ngOnInit() {
  console.log("Sono in ngOnInit di AppComponent");
}

getStatoAnimazione(outlet: RouterOutlet) {
  console.log(outlet.activatedRouteData['statoanim']);
  return outlet.activatedRouteData['statoanim'];
}
}

```

Ogniqualvolta cambierem o pagina, se nella route è specificato un parametro `data`, questo sarà passato alla variabile del template `out`, che a sua volta attiverà la chiamata alla funzione `getStatoAnimazione()` valorizzando così correttamente lo stato dell'animazione “cambioPagina”.

Lo possiamo verificare andando a inserire dei messaggi di log, sia nel componente `AppComponent` che nel componente `HomeComponent`. In particolare li inseriremo nel costruttore, in `ngOnInit`, e nella funzione `getStatoAnimazione()`.

Non appena l'applicazione si avvia per la prima volta e si istanzia il componente `AppComponent` (punto 1 e 2 di figura 16.1), il valore restituito da `getAnimazione()` sarà `undefined` (punto 3). Subito dopo, viene trovata la corrispondenza tra l'url e la prima route, e all'interno di `<router-outlet>` è inserito il componente `treni.component.ts` (punto 4).

Il dato presente nella route nel frattempo è passato alla funzione `getStatoAnimazione()`, che finalmente lo visualizzerà (5). E così via per ogni nuovo componente richiamato non appena si verifica una corrispondenza con la relativa route .

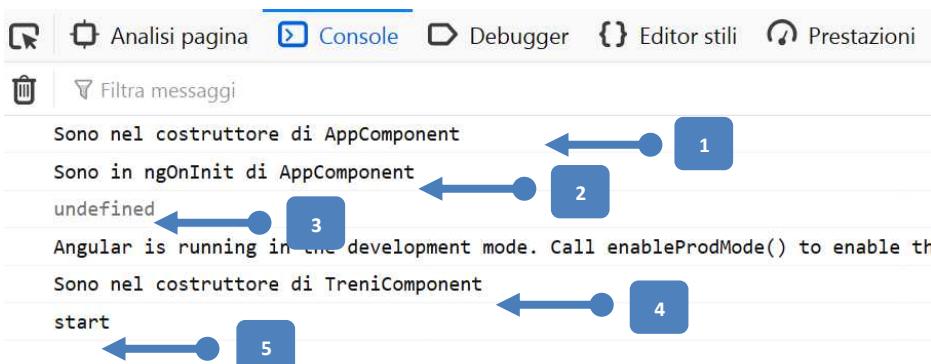


Figura 16.1 - Ciclo di vita del componente `AppComponent`, e `HomeComponent` all'apertura dell'app.

Il passo finale sarà l'aggiunta dell'animazione vera e propria.

16.4 Come definire l'animazione d'ingresso e uscita

Dicevamo all'inizio quali fossero le due difficoltà che dovevamo risolvere. Abbiamo capito come risolvere la prima , grazie a i dati inseriti nel le route e al contenitore padre di `<router-outlet>`. Vediamo come risolvere la seconda, ossia fare in modo che la pagina uscente si sposti verso destra e la nuova pagina entri da sinistra.

Come già fatto per l'animazione di singoli elementi di un componente, dovremo creare l'animazione all'interno di `app.component.ts` e chiamarla `cambioPagina`. Lo sappiamo già fare sfruttando la proprietà `animations` del decoratore `@Component`.

A differenza di quello visto in precedenza per i singoli elementi di una pagina, dove la definizione dello stato ci serviva per creare l'equivalente transizione, in questo caso possiamo applicare gli stessi concetti visti quando abbiamo parlato di animazione d'ingresso e di uscita di un elemento.

Ipotizziamo di animare tutte le pagine allo stesso modo, pertanto il trigger che farà partire l'animazione, non dovrà prendere in considerazione uno stato specifico (es. `start => dettaglio`), ma un qualsiasi stato. Questo il motivo per cui possiamo sfruttare il simbolo speciale dell'asterisco e la bidirezionalità dell'animazione.

```
animations: [
  trigger('cambioPagina', [
    // prendo in considerazione tutti i possibili stati
    transition('* <=> *', [
      ...
    ])
  ])
]
```

Chiaramente se vogliamo rendere unica l'animazione di specifiche pagine, dovremo personalizzarla per ogni cambio pagina, e quindi usare i nomi specifici degli stati coinvolti nella transizione.

Il passo successivo sarà sfruttare le nostre conoscenze sui CSS , sapendo che le due pagine interessate dall'animazione e individuate dai selettori `:enter` e `:leave`, saranno, per un breve periodo di tempo, entrambe presenti nella vista.

Il punto di partenza è applicare un posizionamento assoluto ad entrambe le pagine , in modo che a livello di visualizzazione possano coesistere. Chiaramente dovr emo verificare che tutti gli elementi abbiano un posizionamento relativo .

Per applicare in modo chirurgico uno stile ad un elemento della vista, Angular ci mette a disposizione la funzione `query()`, che agisce proprio come un selettori CSS. La sintassi base è:

```
query('selettore_tag', animate())
```

Nel nostro caso, gli elementi da targettizzare sono individuati dai due selettori predefiniti forniti da Angular ossia `:leave` e `:enter`. Possiamo quindi scrivere:

```
transition('* <=> *', [
  style({ position: 'relative' }),
  // imposto uno stile predefinito in entrata e uscita
  query(':enter, :leave', [
    style({
      position: 'absolute',
      top: 0,
      left: 0,
      width: '100%'
    })
  ])
])
```

A questo punto, dobbiamo procedere con lo spostamento in modo da partire con la pagina in ingresso posizionata fuori dallo schermo verso sinistra . Per questo applicheremo un posizionamento a sinistra del -100% e un'opacità che parte dal valore zero.

```
transition('* <=> *', [
  style({ position: 'relative' }),
  query(':enter, :leave', [
    style({
      position: 'absolute',
      top: 0,
      left: 0,
      width: '100%'
    })
  ]),
  query(':enter', [
    style({ left: '-100%', opacity:0})
  ])
])
```

Ora che abbiamo posizionato correttamente tutti gli attori in gioco, siamo pronti per applicare l'animazione.

L'ipotetica pagina già visibile nello schermo, dovrà essere spostata al di fuori della schermata, verso destra, in modo da far apparire la nuova pagina in arrivo da sinistra.

Abbiamo già visto come si realizza questo effetto. La differenza rispetto a prima è che le animazioni dovranno essere eseguite contemporaneamente all'interno della stessa transizione, e non a intervalli diversi.

Per fare questo Angular ci mette a disposizione una nuova funzione che è `group()`, che come dice lo stesso nome, permette di raggruppare l'insieme delle animazioni da eseguire contemporaneamente sugli elementi indicati.

La sintassi è:

```
group([animazione1, animazione2, ...])
```

Sfruttando ancora la funzione `query()`, possiamo individuare le due pagine in gioco e solo a queste applicare l'animazione con lo stile finale a cui arrivare:

```
transition('* <=> *', [
  // ...
  group([
    query(':leave', [
      animate('250ms ease-out', style({ left: '100%', opacity:0 }))
    ]),
    query(':enter', [
      animate('250ms ease-in', style({ left: '0%', opacity:1 }))
    ])
  ])
])
```

La prima animazione, di durata pari a 250 ms, sposta verso destra l'attuale pagina (`left: '100%`) e imposta l'opacità a “0”, mentre la seconda, sempre di durata pari a 250 ms, fa entrare la nuova pagina posizionandola al posto della precedente (`left: '0%`) con opacità pari a “1”.

Qui sotto puoi vedere il codice completo dell'animazione all'interno del componente `app.component.ts`:

app.component.ts

```

import { Component, OnInit } from '@angular/core';
import { RouterOutlet } from '@angular/router';
import { trigger, state, style, animate, transition, query, group }
    from '@angular/animations';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css'],
  animations: [
    trigger('cambioPagina', [
      transition('start <=> dettaglio', [
        style({ position: 'relative' }),
        query(':enter, :leave', [
          style({
            position: 'absolute',
            top: 0,
            left: 0,
            width: '100%'
          })
        ]),
        query(':enter', [
          style({ left: '-100%', opacity:0 })
        ]),
        group([
          query(':leave', [
            animate('2500ms ease-out', style({ left: '100%', opacity:0 }))
          ]),
          query(':enter', [
            animate('2500ms ease-out', style({ left: '0%', opacity:1 }))
          ])
        ])
      ])
    ])
  ]
})

export class AppComponent implements OnInit {
  title: string;

  constructor() {
    this.title = 'MetroChat con dettaglio';
  }

  getStatoAnimazione(outlet: RouterOutlet) {
    return outlet.activatedRouteData['statoanim'];
  }
}

```

16.5 Spostare il menu al di fuori del template di ogni componente

Come ricorderai dai precedenti capitoli, all'interno di ogni template era stato inserito anche il relativo menu.

Questo perché nella vista di dettaglio avevamo bisogno di cambiare l'elenco dei link visualizzati a favore di una singola freccia per tornare indietro.

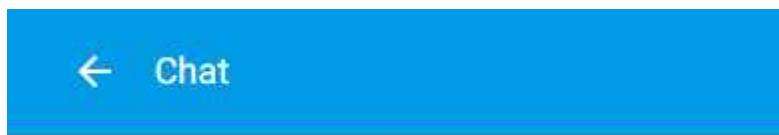


Figura 16.2 – Menu sezione di dettaglio dell'applicazione MetroApp

Ora che abbiamo capito come sfruttare le route per passare ulteriori dati statici, potremmo spostare l'informazione legata al lo stato del menu, direttamente nella route.

In particolare, inseriremo all'interno della route associata alla visualizzazione della pagina di dettaglio, una proprietà di nome `menuback`, valorizzata con la stringa 'ON', in modo da recuperarla e passarla in ingresso al componente `menu.component.ts`.

```
app.module.ts
```

```
// ...
{
  path: 'inarrivo/dettaglio',
  component: DettagliTrenoComponent,
  data: { statoanim: 'dettaglio', menuback: 'ON' },
  children: [...]
}
// ...
```

Poi dobbiamo trasferire il componente `menu.component.ts` presente nei template di ogni pagina grazie al selettore `<menu>`, direttamente nel template del padre di tutti i componenti, ossia `app.component.ts`, in modo che non sia animato con il resto della pagina, ma rimanga fisso nello schermo.

E' chiaro quindi che dovrà essere lasciato esterno al blocco in cui ho inserito il trigger dell'animazione.

app.component.html

```
<ca-menu></ca-menu>
<main [@cambioPagina]="getAnimazione(o)">
    <router-outlet #out = "outlet" ></router-outlet>
</main>
```

E infine dobbiamo aggiungere al menu una proprietà d'ingresso, che ci servirà per passare l'informazione ricevuta dalla route, proprio come fatto per l'animazione di cambio pagina. Potremmo chiamarla con lo stesso nome della variabile d'ingresso usata per il componente *menu.component.ts* visto nel capitolo 10.

app.component.html

```
<ca-menu [menuback]="getMenuBack(o)"></ca-menu>
<main [@cambioPagina]="getAnimazione(o)">
    <router-outlet #out = "outlet" ></router-outlet>
</main>
```

Il metodo `getMenuBack()` recupererà la proprietà `menuback` dalla route, quindi il valore “ON” che, sulla base della logica inserita nel componente *menu.component.ts*, ci permetterà di visualizzare solo la freccia per tornare indietro, invece che tutte le voci del menu.

La relativa funzione sarà molto simile a quella già vista per recuperare il tipo di animazione per il cambio pagina.

```
//...
 getMenuBack(outlet: RouterOutlet) {
    return outlet.activatedRouteData['menuback'];
}
//...
```

Il codice completo del componente *app.component.ts* diventerà:

app.component.ts

```
//...
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  animations: [
    //...
  ]
})
```

(continua)

```

export class AppComponent implements OnInit {
  title: string;

  constructor() {
    this.title = 'MetroChat con dettaglio';
  }

  ngOnInit() {
  }

  getStatoAnimazione(outlet: RouterOutlet) {
    return outlet.activatedRouteData['statoanim'];
  }

  getMenuBack(outlet: RouterOutlet) {
    return outlet.activatedRouteData['menuback'];
  }
}

```

Mentre il template sarà:

```

app.component.html

<ca-menu [menuback]="getMenuBack(o)"></ca-menu>
<main [@cambioPagina]="getAnimazione(o)">
  <router-outlet #out = "outlet" ></router-outlet>
</main>

```

L'altra avvertenza è spostare il template del componente *dettagliotreno.component.ts* all'interno di un file *.html* per non lasciarlo all'interno del corpo della classe.

Questo perché (forse è un bug, ma devo indagare meglio), la presenza di un *<router-outlet>* interno alla proprietà *template* del decoratore *@Component* crea dei problemi all'animazione della pagina d'uscita, quando clicco sulla freccia per tornare alla lista dei treni .

Capitolo 17

Progressive Web Application (PWA)

17.1 Passare da una Web App a una PWA

La quasi totalità dei siti web attuali, è facilmente consultabile da un qualsiasi dispositivo mobile. Molti di questi hanno una grafica e un layout che assomiglia alle classiche applicazioni native scaricabili e installabili nel proprio dispositivo (smartphone e tablet) dai vari store.

Il merito è dovuto ai CSS e all'evoluzione dei browser, che ha permesso di sviluppare i classici "layout responsive" ossia che si adattano alle dimensioni della finestra di visualizzazione.

Tuttavia, fino a qualche anno fa, rispetto alle classiche applicazioni native, rimanevano due sostanziali differenze. La prima era l'impossibilità d'installare un intero sito nel proprio smartphone, come se fosse un'app richiamabile da un'icona, e la seconda, l'impossibilità di rendere navigabile un sito anche se non ha una connessione ad internet.

Dal 2016, a partire dal browser Chrome e poi via via per tutti gli altri, si è potuto ovviare anche a queste due limitazioni, permettendo così allo sviluppatore di sfruttare nuove API.

Oggi, una classica applicazione web (Web App) consultabile con un qualsiasi browser e un qualsiasi dispositivo (desktop, mobile), può essere trasformata in quella che viene chiamata in gergo tecnico Progressive Web App (PWA). Lo stesso nome ci suggerisce che è una "evoluzione" di una normale applicazione web.

E con evoluzione intendo, principalmente, proprio la possibilità di installarla sul un dispositivo mobile o computer fisso (desktop), e di renderla accessibile e funzionante, in parte, anche senza una connessione ad internet (offline).

Questo permette all'utente di continuare a sfruttare quelle applicazioni che non necessitano di una connessione sempre attiva (realtime), cosa non possibile con un normale sito web.

Il fatto poi che sia installabile, permetterà all'utente di accedervi tramite un'icona facilmente richiamabile , potendo così interagire più frequentemente. Questo aspetto è anch'esso importante perché in questo modo si crea quel rapporto di fidelizzazione molto usato dagli esperti di marketing per la vendita di prodotti e servizi.

La sola presenza di queste due caratteristiche permette di posizionare le PWA a un livello sempre più vicino alle applicazioni native, senza l'indubbio svantaggio di dover imparare un nuovo linguaggio di programmazione (vedi Java/Kotlin per Android e Swift per iOS).

Per identificare i pregi delle PWA, spesso si usa l'acronimo F.I.R.E (Fast, installable, reliable, engaging).

La scelta quindi di passare da una semplice web app a una PWA , deve essere fatta con la consapevolezza di voler sfruttare questi vantaggi. Inoltre ogni applicazione web consultabile con un browser, può essere trasformata in PWA e non solo le applicazioni sviluppate con Angular.

Tuttavia, grazie ad Angular CLI, questa trasformazione risulta molto semplice, perché sarà sufficiente installare una libreria creata appositamente per questo scopo, e poi procedere con una serie di configurazioni e personalizzazioni al fine di ottenere il risultato desiderato.

Chiaramente i vantaggi di una PWA non si limitano alla possibilità d'installazione e navigazione offline. Una PWA, così come una Web App è a tutti gli effetti un classico sito web, quindi con le tecniche SEO è possibile indicizzare più pagine sui motori di ricerca, ognuna per specifiche parole chiave cercate dall'utente, cosa che per le applicazioni native risulta difficile da fare.

Inoltre si possono implementare funzionalità che un tempo era impensabile poter avere sul web, come l'invio e la ricezione di notifiche push, la geolocalizzazione, l'accesso alla fotocamera, bluetooth, ecc.

Le PWA sono sicure, perché tutte le chiamate effettuate all'esterno del dispositivo, sono fatte tramite il protocollo HTTPS e gestite da un file, chiamato service worker, che funge da proxy, intercettando tutte le chiamate in uscita e in ingresso all'applicazione.

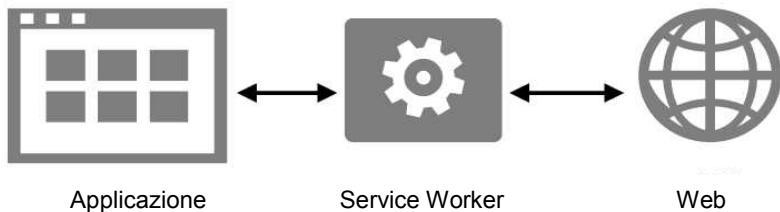


Figura 17.1 – Azione del service worker in una PWA

E’ proprio quest’ultimo file che garantisce minori chiamate alla rete con conseguente maggior velocità d’ esecuzione dell’applicazione, potendo recuperare alcune risorse direttamente dalla cache del browser .

L’altro componente fondamentale è il file chiamato *manifest*, che rende possibile l’installazione di una Web App nel dispositivo dell’utente, permettendo così l’accesso tramite un’icona, esattamente come se fosse un’applicazione nativa .

A dire il vero, nel corso degli ultimi anni sono emerse delle problematiche in termini di privacy , che hanno costretto alcuni produttori di browser a limitarne le funzionalità (vedi Safari e Firefox in primis). Non è detto che in futuro si arrivi a una totale rivisitazione del concetto di PWA, ma per ora, sfruttiamo quello che ci è concesso di fare.

17.2 Cos’è un Service Worker e perché si usa

Nella sua forma più semplice, un *service worker* è uno script che viene installato ed eseguito nel browser dell’utente, in un *thread* separato da quello in cui è eseguito il codice JavaScript della pagina . Questo garantisce che non sia bloccante per l’applicazione e pericoloso, in quanto non ha accesso al DOM . Ha un proprio ciclo di vita, che vedremo in seguito e fa uso massiccio delle *promise*, poiché generalmente aspetteremo che determinati eventi si verifichino, al fine di prendere delle decisioni.

In primo luogo permette di intercettare e modificare la navigazione e le richieste di risorse dalla rete, per darci il completo controllo su come si deve comportare l’applicazione in determinate situazioni. La più ovvia è quando la connessione di rete non è disponibile.

L’utilizzo di un service worker, se opportunamente configurato, riduce la dipendenza dell’applicazione dalla rete e può migliorare notevolmente l’esperienza di navigazione dell’utente in quanto può sfruttare l’accesso alla cache locale. In

questo modo l'applicazione può funzionare anche in assenza di connessione di rete.

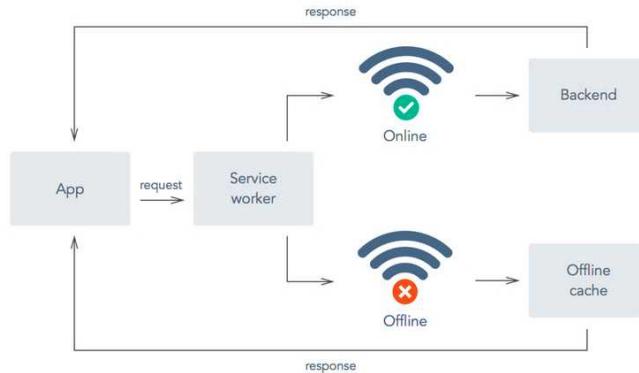


Figura 17.2 – Processo di richiesta di una risorsa di rete in presenza di un service worker

Con opportune configurazioni possiamo decidere quali risorse provenienti dalla rete memorizzare nella cache e quali no. Il proxy infatti non è limitato alle richieste effettuate API programmatiche, ma include anche le richieste a risorse statiche come file HTML, CSS, JavaScript.

Il service worker viene conservato nel dispositivo dell'utente anche dopo che quest'ultimo ha chiuso il browser. Il browser, ad ogni accesso dell'utente, verificherà sul server eventuali aggiornamenti e in presenza di una differenza rispetto alla versione installata in locale (anche un solo byte), il browser reinstallerà la nuova versione in background.

Come dicevamo viene eseguito su connessioni HTTP, con la sola eccezione dello sviluppo in locale per cui non è richiesta.

La cosa interessante da sottolineare è che può essere implementato su un qualsiasi sito web, indipendentemente dal framework con cui è realizzato.

Il metodo più rapido per capire se un sito web ha le funzionalità di un PWA, è verificare proprio l'esistenza di questo file. Possiamo farlo accedendo al pannello “ispezione risorse” di Chrome e poi spostandosi nella sezione “Application” e selezionando “Service Workers” (fig.17.3). Tipicamente il file ha nome `ngsw.js`, ma può essere chiamato in qualsiasi modo, purché sia un file JavaScript.

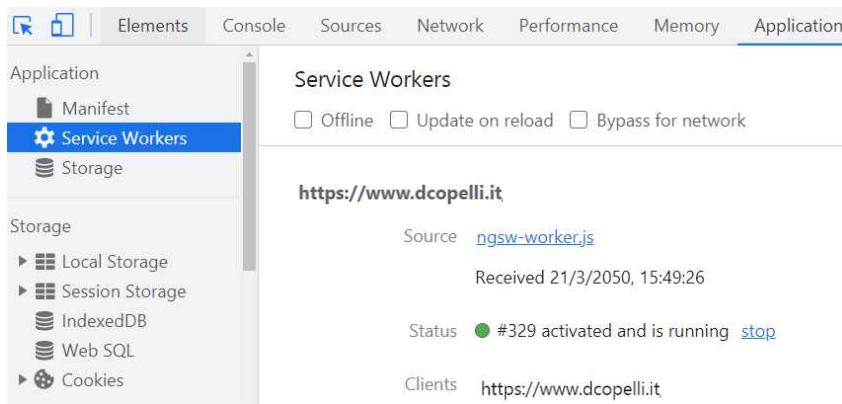


Figura 17.3 – Visualizzazione del service worker memorizzato in Chrome

L’altro dato interessante è lo “stato”, in quanto, come dicevamo prima, l’installazione segue un preciso ciclo di vita. Giusto per non appesantire in questa fase la trattazione, rimanderemo questi aspetti a una successiva sezione.

Le specifiche dell’API Service Worker sono molto vaste. Noi tratteremo solo i principali utilizzati, in relazione a quello che ci offre Angular CLI. Per casi più sofisticati, sarà necessario creare dei service worker personalizzati. Maggiori dettagli li puoi trovare nel link presente nella sezione “Sitografia” alla fine del libro.

17.3 Usare Angular CLI per creare una PWA

In Angular, il processo per trasformare un’applicazione web in una PWA, richiede pochi semplici passaggi se ci avvaliamo delle librerie sviluppate per Angular, in particolare di `@angular/pwa`. Il primo passo è quello di spostarci nella cartella principale del progetto, e nella finestra dei comandi, inserire:

```
ng add @angular/pwa
```

Con questa riga, saranno eseguite una serie di operazioni, che se eseguite singolarmente, richiederebbero diversi minuti. Vediamo le ad una ad una cercando di dare una prima sommaria spiegazione del significato.

- 1) Viene installato `@angular/service-worker` all’interno del progetto, modificando così il file `package.json`.

2) All'interno della cartella principale del progetto, viene creato il file di configurazione del service worker, con nome `ngsw-config.json` che ci servirà per indicare quali risorse di rete, dovranno essere memorizzate nella cache del browser del dispositivo dell'utente (fase di installazione). Questo file, cambia nome in `ngsw.json`, non appena creiamo la versione di produzione dell'app (comando `ng build` visto all'inizio del libro).

3) Affinché il browser possa installare il file `ngsw-config.js`, dobbiamo inserire una riga per iniettare il codice che effettuerà l'effettiva registrazione del service worker. All'interno di `app.module.ts` comparirà la seguente riga:

app.module.ts

```
//...
import { ServiceWorkerModule } from '@angular/service-worker';
//...
@NgModule({
//...
imports: [
//...
ServiceWorkerModule.register('ngsw-worker.js', {
    enabled: environment.production
})
]
})
```

4) Nella cartella `src` del progetto, viene creato un nuovo file, di nome `manifest.webmanifest`, che fornirà al browser tutte le informazioni per mostrare l'applicazione, non appena sarà installata nel dispositivo dell'utente.

5) Affinché quest'ultimo file sia visibile all'applicazione, è necessario aggiungerlo all'interno del meta tag `<head>` della pagina `index.html`

index.html

```
<!doctype html>
<html>
<head>
//...
<link rel="manifest" href="manifest.webmanifest">
</head>
//...
```

6) Per gestire il colore di sfondo della barra di stato del browser nei dispositivi mobili e nelle finestre del desktop , sempre all'interno del meta tag `<head>` della

pagina *index.html*, si aggiungono le informazioni sul codice colore tramite l'attributo `theme-color`.

7) Nella cartella `src` del progetto, viene creata una nuova cartella di nome `assets` che conterrà l'icona (alle diverse risoluzioni) che ci servirà per identificare e lanciare l'applicazione sia da dispositivi mobili che desktop.

Giunti a questo punto, l'applicazione è a tutti gli effetti una PWA.

Una parte di queste operazioni, sono le stesse che dovrai fare per trasformare in PWA, una qualsiasi applicazione creata senza sfruttare Angular.

Chiaramente non abbiamo ancora effettuato alcuna personalizzazione, ma il tutto è già funzionante. E' necessario proseguire nel processo di configurazione dei diversi elementi creati, e per farlo, conviene impostare un server locale al fine di eseguire tutti i test.

17.4 Creare la versione di produzione e testare il service worker

All'inizio del libro abbiamo visto come si crea la versione di produzione dell'applicazione da trasferire nel web server.

Al fine di testare più velocemente – in locale – le nuove funzionalità di cui abbiamo dotato l'applicazione, è necessario installare un server HTTP.

Uno dei tanti che puoi trovare online è il package `http-server` che possiamo installare, a livello globale, sfruttando la seguente riga di comando:

```
npm install --global http-server
```

La riga per far partire il server è un po' prolissa, perché contiene i parametri opzionali per indicare la porta in cui servire le richieste alle risorse di rete, e quello per eliminare le informazioni sul caching delle risorse, trasmesse tramite le intestazioni HTTP.

Dovrai anche fornire l'informazione sul percorso dove si trovano i file creati con il comando `ng prod` (nel nostro caso `dist/MetroChat`).

All'interno del file `package.json`, in corrispondenza alla chiave `script`, possiamo quindi scrivere :

```
package.json

"script": {
  ...
  "prod": "ng build --prod && http-server -c-1 -p 8080 dist/MetroChat"
}
```

Abbiamo inserito la riga direttamente nel comando che crea la versione di produzione dell'applicazione, in modo da non doverla digitare ogni volta durante i vari test. In questo modo sarà richiamata in automatico con il comando :

```
npm run prod
```

e collegandoci al link `http://localhost:8080`, saremo in grado di visualizzare l'applicazione finale di produzione. L'elenco dei file generati è visibile nella fig.17.4

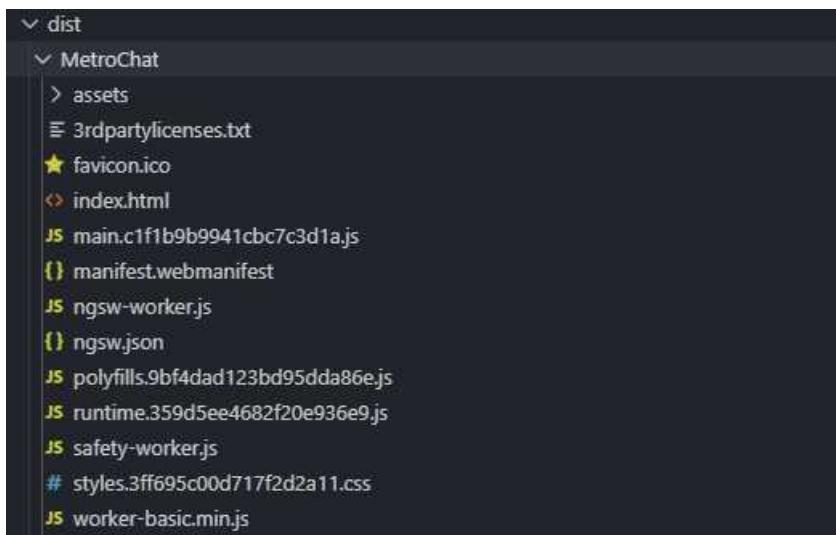


Figura 17.4 – Elenco file nell'app di produzione dopo l'aggiunta del service worker

Nota come per alcuni sia stata inserita una sequenza di caratteri e numeri (hash) che permette di identificare in modo univoco ogni nuova versione di produzione .

Per verificare se il service worker è stato installato ed è funzionante, è necessario aprire la finestra d'ispezione risorse del browser e spostarsi nel menu “Application”. Troveremo una situazione simile a quella di fig. 17.3.

Per testarne il funzionamento, possiamo disabilitare la connessione ad internet e provare a ricaricare la pagina. Con un normale sito web, vedresti la classica pagina d'errore del browser, che informa l'utente dell'assenza di connessione (fig. 17.5).

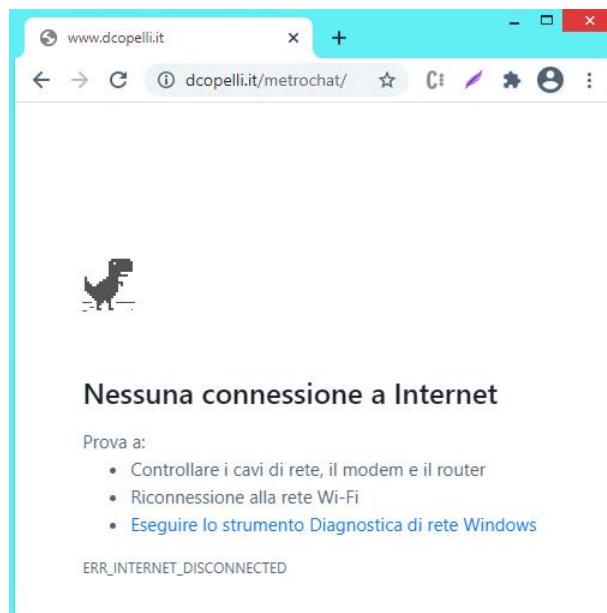


Figura 17.5 – Pagina del browser in assenza di connessione

Nel nostro caso, continueremo a vedere gran parte della grafica, il che significa che una parte dell'applicazione è stata memorizzata nella cache e servita dal service worker (fig. 17.6). Solo la sezione di recupero dati dall'API, non viene visualizzata, ma vedremo più avanti come memorizzare anche questi.

Chiaramente nel caso di un'app realtime come la nostra, avrebbe poco senso mostrare dei dati non aggiornati, ma giocando sulla durata di validità delle risorse nella cache, possiamo comunque fornire un'esperienza utente sicuramente migliore di quella di fig. 17.5.



Treni in Arrivo a questa stazione

Qui sotto puoi vedere l'elenco dei treni in arrivo presso questa ipotetica stazione.

Loading...

Http failure response for
<https://www.dcopelli.it/test/angular/metro/>: 504
 Gateway Timeout

Treni appena partiti v1

Qui sotto puoi vedere la lista dei treni che sono appena partiti e che hai perso.

Figura 17.6 – Pagina d’ingresso dell’app in assenza di connessione internet

Ulteriore conferma dell’azione svolta dal service worker, l’abbiamo se andiamo a verificare da dove sono recuperate le risorse che compongono le voci del menu, il colore, l’icona (fig. 17.7)

Network					
	Name	Status	Type	Initiator	Size
●	material-icon.css	200	stylesheet	inarrivo	(ServiceWorker)
●	materialize.min.css	200	stylesheet	inarrivo	(ServiceWorker)
●	style.css	200	stylesheet	inarrivo	(ServiceWorker)
●	jquery.min.js	200	script	inarrivo	(ServiceWorker)
●	styles.3ff695c00d717f2...	200	stylesheet	inarrivo	(ServiceWorker)
○	materialize.min.css	200	fetch	ngsw-worker.js...	(disk cache)
○	style.css	200	fetch	ngsw-worker.js...	(disk cache)

Figura 17.7 – Parte delle risorse di rete gestite dal service worker

Capiremo in dettaglio come gestire le risorse da memorizzare nella cache , quando parleremo di configurazione del service worker.

Altra cosa interessante da testare è vedere cosa succede se apportiamo una qualsiasi modifica all'applicazione, ad esempio , se cambiamo il testo del menu di accesso da “Login” in “Entra”? Riavviando il server e ricreando la versione di produzione - quindi un nuovo file *ngsw.json* - non noteremmo alcuna modifica.

Se invece aggiorniamo la pagina, allora riusciamo a vederla. Questo comportamento bizzarro è in realtà voluto dagli sviluppatori di Angular, e ha a che fare con il ciclo di vita di un service worker (registrazione, scaricamento, attivazione).

Vedremo meglio questi concetti successivamente, ma giusto per non lasciarti a bocca asciutta, il motivo è legato al fatto che la prima volta che un utente accede al link di una PWA, il browser scarica e installa in locale il service worker, che entra in funzione (attivazione) al successivo accesso dell'utente o in seguito a un refresh della pagina del browser . (fig. 17.7)

A questo punto , ad ogni nuovo accesso, il browser controlla la presenza di una nuova versione del file *ngsw.js* e se la trova, la scarica ma non l'attiva, ossia non la sostituisce subito al vecchio service worker. Per quest'ultimo passaggio è necessario che l'utente chiuda tutte le finestre del browser in cui è presente l'applicazione.

Il motivo di questa scelta è legato principalmente a un discorso di coerenza, nel caso in cui l'utente abbia più finestre aperte con la stessa app. Ogni finestra dovrà visualizzare la stessa versione.

E' chiaro quindi che diventa fondamentale avere un meccanismo che segnali la presenza di nuove versioni dell'applicazione, nel caso l'utente si trovi in una delle situazioni precedenti. Anche questo lo tratteremo in seguito.

Per ora preoccupiamoci di capire come apportare le personalizzazioni di base sia al file *manifest.webmanifest* che al service worker, per soddisfare alle due caratteristiche evidenziate in precedenza: installazione e navigazione offline.

17.5 Rendere l'app installabile: il file App Manifest

Se per i dispositivi mobili e fissi, esistono opportuni store da cui scaricare le applicazioni, come è possibile far installare un'applicazione che risiede su uno spazio web privato ?

Se navighi spesso con il tuo smartphone, sicuramente ti sarà capitato di vedere comparire un messaggio in pop-up simile a quello di fig.17.8.

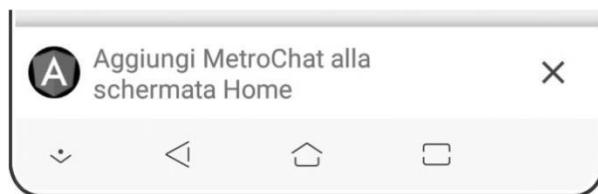


Figura 17.8 – Messaggio che invita l’utente a installare l’applicazione nei dispositivi mobili

Ebbene, questo è il punto d’ingresso per permettere l’installazione di una Web App nel dispositivo dell’utente.

Una volta installata, comparirà un’icona sulla schermata “home” del dispositivo. Analogico discorso per i dispositivi fissi se usi Chrome o Safari, solo che dovrà cliccare sull’apposito simbolo presente nella barra degli indirizzi (fig. 17.9).

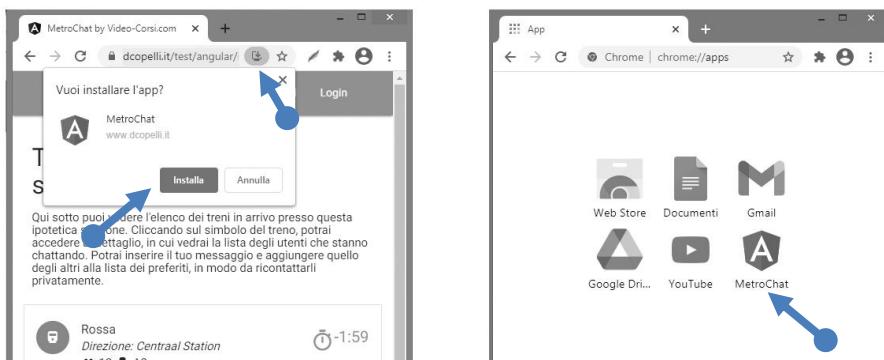


Figura 17.9 – Pop-up di installazione applicazione su Chrome e icona sulla schermata app

Tutto questo è possibile grazie al file *manifest.webmanifest* installato in precedenza con Angular CLI.

Il formato del documento è il classico JSON ed ha estensione *webmanifest*, quella suggerita dalle specifiche dello standard.

In realtà i browser sono in grado di interpretare correttamente ogni file con estensione *.json*, quindi anche il classico *manifest.json*, a patto chiaramente di aver inserito nelle intestazioni della pagina *index.html*, il riferimento al file.

I dati che inseriremo all'interno, comunicheranno al browser come dovrà comportarsi la PWA, una volta installata sul desktop di un computer fisso o sulla schermata “home” del dispositivo mobile dell'utente.

Gran parte dei dati sono già stati inseriti di default all'interno di una sequenza di proprietà. Dovremo solo personalizzarli.

```
src/manifest.webmanifest

{
  "name": "MetroChat",
  "short_name": "MetroChat",
  "theme_color": "#1976d2",
  "background_color": "#fafafa",
  "display": "standalone",
  "scope": "./",
  "start_url": "./",
  "icons": [
    {
      "src": "assets/icons/icon-72x72.png",
      "sizes": "72x72",
      "type": "image/png",
      "purpose": "maskable any"
    },
    //...
  ]
}
```

Qui sotto trovi il significato delle proprietà più importanti:

name: il testo rappresentativo del nome dell'app, che sarà visualizzato, ad ogni accesso, nella schermata di benvenuto. Quest'ultima viene chiamata “splash screen” – fig.17.10 ed è visibile solo nei dispositivi mobili.

short_name: è il nome usato in alternativa a *name*, in mancanza di spazio. Tipicamente compare sia nel messaggio che inviterà l'utente a installare l'app (fig. 17.8), sia nel messaggio di pop-up, sia sotto l'icona associata al lancio dell'applicazione.

icons: le icone, alle diverse risoluzioni, che permetteranno all'utente di individuare e aprire l'app su un dispositivo mobile o su un computer. Dovranno chiaramente essere tutte uguali a livello di grafica e cambiare solo in risoluzione. Le dovrà memorizzare all'interno della cartella *src/assets*.

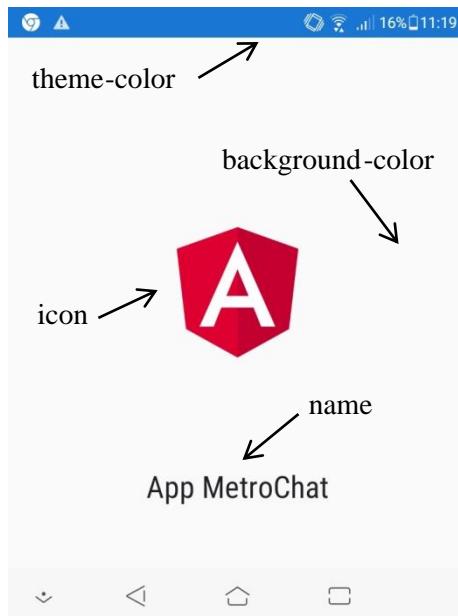


Figura 17.10 – Schermata di benvenuto (splash screen) nei dispositivi mobili

Un utile link per crearle tutte , partendo da un'unica immagine quadrata di almeno 512px di larghezza, è usare uno dei tanti servizi online, ad esempio:

<https://www.favicon-generator.org/>



Figura 17.11 – Servizio per generare le icone alle diverse dimensioni

display: la modalità di visualizzazione della finestra contenitore dell'applicazione. Il valore tipico è `standalone` al fine di visualizzare l'app senza gli elementi tipici di una finestra del browser, come la barra degli indirizzi o altri menu .

Questo è il valore da preferire soprattutto se l'applicazione sarà frutta tramite dispositivi mobili , in quanto permette di avvicinarsi alla grafica di un'app nativa.

scope: la stringa che rappresenta l'ambito di validità dell'insieme degli URL a cui l'utente può navigare dall'applicazione. Tutti gli URL che non appartengono a questo ambito, non saranno gestiti con le regole presenti nel file *manifest.webmanifest*, ma con le regole predefinite del browser e visualizzati in finestre diverse da quelle dell'app. Solitamente coincide con l'origine del dominio da cui si sta installando l'applicazione “./” (occhio allo slash finale), ma potrebbe includere anche una sottocartella (es. “./angular/”).

start_url: è il percorso su cui il browser dell'utente sarà indirizzato non appena si clicca sull'icona dell'applicazione. Spesso comprende anche il nome del file specifico da lanciare e potrebbe contenere dei parametri di tracking (es. index.html?medium=pwa). Può essere indicato in modo assoluto (compreso di https) oppure relativo. In quest'ultimo caso la base di partenza è l'URL dove risiede il file *manifest.webmanifest*. A livello di privacy utente potrebbero esserci degli abusi se il programmatore usasse questo parametro per identificare in modo univoco l'utente.

theme_color: i colori da associare alla barra di stato e degli strumenti. Questo dato sarà iniettato nel meta tag `<head>` della pagina *index.html*. Spesso il valore si sceglie sulla base del colore di sfondo del menu dell'app.

background_color: i colori del la prima videata che l'utente vedrà nei dispositivi mobili, per pochi secondi (splash screen), all'apertura dell'app. Solo Chrome attualmente fa vedere questa schermata iniziale. Fig. 17.10.

17.6 Configurare il service worker

Abbiamo imparato che il servizio worker è il centro di controllo tramite il quale decidiamo le politiche di riempimento della cache. Il browser, ad ogni accesso da parte dell'utente, dovrà controllare il file per verificare se ci sono aggiornamenti, in seguito ad esempio a novità nella grafica, nuovi servizi, o correzione di bug.

Ogni modifica al codice dell'applicazione, comporta la creazione di un nuova versione di tale file, quindi all'interno del browser dell'utente potrebbero coesistere due versioni: una installata la prima volta che l'utente ha avuto accesso all'applicazione, e una in attesa di attivazione, legata a un aggiornamento.

Come dicevamo in precedenza, a fine di rendere attivo un aggiornamento, è necessario eseguire un'operazione di refresh della pagina, o di chiusura di tutte le finestre in cui è presente un'istanza del precedente service worker.

Tra le risorse obbligatorie da memorizzare nella cache, ci sono chiaramente quelle necessarie al funzionamento dell'app ossia il file *index.html* e tutti i file CSS e JavaScript.

Le personalizzazioni saranno tutte inserite all'interno del file *ngsw-config.json*. Ti ricordo che quest'ultimo, durante la creazione del progetto di produzione, viene rinominato in *ngsw.json* e salvato nella relativa cartella radice.

```
ngsw-config.json

{
  "$schema": "./node_modules/@angular/service-worker/config/schema.json",
  "index": "/index.html",
  "assetGroups": [
    {
      "name": "app",
      "installMode": "prefetch",
      "resources": {
        "files": [
          "/favicon.ico",
          "/index.html",
          "/manifest.webmanifest",
          "/*.css",
          "/*.js"
        ]
      }
    },
    {
      "name": "assets",
      "installMode": "lazy",
      "updateMode": "prefetch",
      "resources": {
        "files": [
          "/assets/**",
          "/*.(eot|svg|curl|jpg|png|webp|gif|otf|ttf|woff|woff2|ani)"
        ]
      }
    }
  ],
  "dataGroups": []
}
```

Grazie ad Angular CLI, sono già state valorizzate in automatico una serie di proprietà, con i riferimenti ai file e ai gruppi di file che rappresentano proprio tutte le risorse presenti nella cartella principale del progetto di produzione.

Nella fig.17.12, puoi vedere il risultato che si ottiene ispezionando la Cache Storage del browser con lo strumento “Ispeziona” .

#	Name	Response-Type...	Content-Type	C...
0	/favicon.ico	basic	image/x-icon	
1	/index.html	basic	text/html	
2	/main.541f270d50d928fedcda.js	basic	application/...	
3	/manifest.webmanifest	basic	text/plain	
4	/polyfills.8c151b8375e767ff858f.js	basic	application/...	
5	/runtime.0e49e2b53282f40c8925.js	basic	application/...	
6	/styles.3ff695c00d717f2d2a11.css	basic	text/css	

Figura 17.1.2 – Elenco dei file memorizzati nella Cache Storage

Le due sezioni più importanti di `ngsw-config.json`, sono `assetGroups` e `dataGroup` (quest’ultima non sempre è inserita a livello predefinito da Angular CLI). Entrambe richiedono le due proprietà `name` e `files` per raggruppare le risorse a cui applicare la stessa politica di installazione (`installMode`) e aggiornamento (`updateMode`).

Ogni elemento dell’array indicato nella chiave `files`, deve iniziare con il simbolo “`/`” e il percorso di recupero farà riferimento alla cartella principale del progetto di produzione.

E’ possibile sfruttare anche le Regular Expression per identificare un insieme di file che corrispondono a certi requisiti (es. tutti i file CSS, tutti i file JavaScript ecc.).

L’ordine in cui sono inserite le risorse nei vari gruppi, è importante in quanto non appena si individua una corrispondenza, tutte le successive sono ignorate.

Inoltre è possibile specificare quando dovranno essere scaricate, tramite la chiave `installMode`. Il valore predefinito è “`prefetch`” che richiede più spreco di banda, ma permette di visualizzare gran parte dell’applicazione anche in assenza di connessione dati in quanto tutte le risorse sono caricate al primo accesso.

Per le immagini non indispensabili, si preferisce usare la modalità `lazy` che permetterà di scaricare le risorse solo se richieste dall’utente in seguito al caricamento di una specifica pagina.

E’ bene ricordare che il service worker si installa dopo che tutta l’applicazione è stata visualizzata dall’utente, quindi anche tutte queste risorse saranno installate in background, senza che l’utente si accorga di un qualche rallentamento nell’utilizzo dell’intera applicazione.

Solo al secondo caricamento dell'intera applicazione, se le risorse sono disponibili nella cache, saranno visualizzate, bypassando così la rete e velocizzando il caricamento dell'app.

Chiaramente, oltre a queste due sezioni, ne esistono delle altre che puoi consultare a questo indirizzo:

<https://developer.mozilla.org/en-US/docs/Web/Manifest>

La proprietà `dataGroups` non sempre è inserita a livello predefinito perché definisce le politiche di cache per risorse tipicamente provenienti da API o CDN.

Nel caso dell'applicazione MetroChat, tutti i dati dei treni in partenza e i messaggi nella chat di ogni treno, devo essere sempre visualizzati in tempo reale, quindi non avrebbe molto senso recuperarli dalla cache.

Eventualmente si potrebbe prevedere di memorizzare sempre una copia delle ultime informazioni recuperate, per ovviare a piccole interruzioni di rete legate, ad esempio, al passaggio del treno all'interno di una galleria senza ripetitori.

Come per la precedente proprietà, anche questa richiede un nome obbligatorio, un insieme di URL che rappresentano le risorse da memorizzare (chiamate alle API), e infine le politiche di conservazione nella cache del browser.

Se volessimo applicarla comunque all'applicazione MetroChat (solo come esercitazione), potremmo scrivere:

```
//...
"dataGroups": [
  {
    "name": "treni-api",
    "urls": [
      "/treni/*",
      // altri endpoint
    ],
    "cacheConfig": {
      "strategy": "freshness",
      "timeout": "6s",
      "maxAge": "1d12h",
      "maxSize": 100
    }
  }
]
```

All'interno della chiave `urls`, abbiamo specificato l'insieme degli endpoint (sotto forma di array di stringhe) a cui applicare le strategie di conservazione elencate all'interno di `cacheConfig`. È possibile creare più gruppi, ognuno dei quali dovrà avere una proprietà `name`, un insieme di endpoint e una relativa politica di conservazione.

Nel nostro caso, abbiamo inserito solo l'endpoint che recupera la lista dei treni. La chiave `strategy` può assumere due valori: `freshness` o `performance`. Con la prima indichiamo al browser di interrogare sempre e comunque la rete e solo se il server non risponde entro uno tempo impostato con `timeout`, recupera dalla cache i dati precedentemente memorizzati.

E' chiaro che in questo modo il browser salverà sempre una copia nella cache, ma la sfrutterà solo se strettamente necessario, favorendo il recupero di dati "freschi" dalla rete.

Con `timeout` specifichiamo per quanto tempo il service worker rimarrà in attesa di una risposta dal server, prima di passare alla cache. Il valore può essere impostato a giorni (`d`), ore (`h`), minuti (`m`), secondi (`s`) e millesimi di secondo (`u`).

Il valore di `maxAge` ci permette di decidere dopo quanto ricontattare il server remoto per aggiornare i dati nella cache. Può anch'esso essere impostato a giorni (`d`), ore (`h`), minuti (`m`), secondi (`s`) e millesimi di secondo (`u`). Nel nostro caso lo abbiamo impostato con scadenza a un giorno (`1d`) e mezzo (`12h`). In pratica, superato questo tempo, alla prima occasione i dati nella cache sono aggiornati.

L'altra opzione, che è quella di `default`, permette di sfruttare sempre la cache come prima risorsa per il recupero dei dati, e di affidarsi al parametro `maxAge`, per decidere dopo quanto ricontattare il server remoto per aggiornare i dati nella cache.

Una strategia che riduce il numero di richieste da inviare in rete e che spesso si usa per dati che non devono essere aggiornati frequentemente. In questo caso non è necessario impostare il parametro `timeout`.

Con `maxSize` impostiamo invece il massimo numero di richieste da memorizzare nella cache.

17.7 Informare l'utente di un nuovo aggiornamento

Abbiamo detto che, non appena l'utente fa un nuovo accesso all'applicazione, il browser controlla la presenza di nuove versioni del service worker, al fine di

scaricarle e attivarle seguendo il classico ciclo di vita accennato in precedenza. L'intervallo temporale tra due controlli è in genere non inferiore a un giorno.

La domanda che potresti farti è cosa succede nel caso tu metta in produzione una versione che presenta un bug e l'utente non chiuda mai l'applicazione o non effettui il do ppio refresh. In questi casi il bug rimane, a meno che non si coinvolga più attivamente l'utente, mostrando esplicitamente un messaggio che lo inviti ad aggiornare l'applicazione.

Per ottenere questo, si usa il service `SwUpdate` presente nella libreria installata all'inizio del capitolo, e in particolare la proprietà `available`, che restituisce un Observable di tipo `UpdateAvailableEvent`. Al verificarsi di questo evento, la nuova versione dell'applicazione – e quindi del servizio worker – è disponibile nel browser dell'utente, così come le eventuali nuove risorse.

Possiamo così segnalarlo all'utente con un messaggio e un bottone che consenta di effettuare il refresh della pagina del browser.

Il tutto dovrà essere fatto all'interno del componente principale `app.component.ts`, perché il service worker si installa al più tardi dopo che tutta l'app si è caricata.

Un possibile codice potrebbe essere:

```
app.component.ts

import { SwUpdate } from '@angular/service-worker';
//...
export class AppComponent {
//...
constructor(private swUp: SwUpdate) {
//...
}
ngOnInit() {
//...
if (this.swUp.isEnabled) {
this.swUp.available.subscribe(() => {
if(confirm("Nuova versione disponibile. Aggiorna")) {
window.location.reload();
}
});
}
}
}
```

Come puoi osservare, abbiamo importato la classe `SwUpdate` dal package `@angular/service-worker`. Con la proprietà `isEnabled`, effettuiamo il

controllo per verificare se il service worker è disponibile nel dispositivo dell'utente.

L'effetto del refresh della pagina, comporta l'attivazione della nuova versione installata in background.

L'argomento PWA è molto vasto e in continuo sviluppo, ma già queste prime informazioni ti permetteranno di fare un deciso passo in avanti per far assomigliare la tua applicazione web a un'applicazione nativa.

17.8 Ciclo di vita di una PWA con service worker

Ora che abbiamo scoperto le principali caratteristiche di un a PWA, proviamo ad analizzare più in dettaglio come si programmano le fasi che contraddistinguono il ciclo di vita di un service worker. Questo sarà utile per creare delle versioni personalizzate a seconda delle esigenze che matureranno nel tempo.

Il ciclo di vita di ogni PWA si può riassumere in una sequenza di azioni che si succedono non appena l'utente accede per la prima volta all'URL dove si trova una PWA (fig. 17.11). Si possono individuare due fasi importanti che sono la registrazione/installazione e l'attivazione.



Figura 17.13 – Ciclo di vita del service worker

Nel caso di applicazioni Angular che sfruttano la libreria `@angular/pwa`, non dobbiamo preoccuparci di programmarle, in quanto sono già incorporate nel codice che viene generato.

Ricorda però che queste applicazioni, risultano avere un comportamento leggermente diverso da quelle che presentano un service worker personalizzato, soprattutto in relazione alla fase di attivazione.

Per spiegare questi concetti, sfrutteremo un'applicazione base creata con le tecniche viste nel primo capitolo. Partiamo subito con il creare un file vuoto – `sw.js` – che rappresenterà il service worker dell'applicazione. Lo memorizziamo nella cartella radice del progetto.

La prima cosa da verificare è che il browser supporti le funzionalità di un service worker. Ci viene in aiuto l'oggetto `navigator` del browser, che dovrà contenere la proprietà `serviceWorker`. Spostiamoci all'interno del file `main.ts` e inseriamo le righe, dopo che l'applicazione è stata interamente caricata nel client.

```
main.ts
//...
platformBrowserDynamic().bootstrapModule(AppModule)
  .then(() => {
    if ('serviceWorker' in navigator) {
      navigator.serviceWorker.register('/sw.js', {
        scope: '/'
      })
      .then(registration => {
        console.log('Service worker registrato');
      });
    }
  });
}) ;
```

Se è compatibile, sfrutteremo l'oggetto `ServiceWorkerContainer` restituito da `navigator.serviceWorker` per accedere al metodo `register` che ci permetterà di installare il service worker con nome `sw.js` e le relative risorse.

Il parametro `scope` è uno dei diversi parametri di configurazione che possiamo usare. Si usa per indicare al service worker di entrare in azione per intercettare tutte le richieste alle risorse del dominio in cui è presente l'applicazione.

L'installazione effettiva può essere monitorata con un gestore d'evento simile a quelli che si usano in JavaScript: l'evento `install`.

sw.js

```

console.log("Installo il service worker");

self.addEventListener('install', e => e.waitUntil(installoRes()));

async function installoRes() {
  console.log("Ora Installo le risorse")
  // sfrutto l'API Cache Storage per memorizzare le risorse
}

```

Installato il file `sw.js`, è necessario sfruttare l'API Cache Storage per memorizzare le risorse nella cache. Al termine, queste saranno disponibili, ma non ancora utilizzabili.

E' necessario infatti un ulteriore step, rappresentato dall'evento `activate` che ha lo scopo di ripulire la cache da eventuali risorse non più utili.

Nel corso degli anni, Chrome ha cambiato il comportamento predefinito riferito a quest'ultima fase e "attualmente" avviene subito e senza bisogno di aggiornare la pagina. Ho scritto "attualmente" tra virgolette perché potrei essere smentito tra un giorno.

sw.js

```

console.log("Installo il service worker");

self.addEventListener('install', e => e.waitUntil(installoRes()));

async function installoRes() {
  console.log("Ora Installo le risorse");
  // sfrutto l'API Cache Storage per memorizzare le risorse
}

self.addEventListener('activate', e => e.respondWith(attivoSW(e)));

async function attivoSW(e) {
  console.log("Attivo il Service Worker");
  // elimino dalla cache le risorse che non servono più
}
}

```

Testiamo l'applicazione per vedere in che sequenza i messaggi sono mostrati nella console. Dalla fig.17.14 notiamo come l'attivazione avvenga dopo la registrazione. Da questo momento in poi, tutte le richieste fatte dall'applicazione a risorse di rete (immagini, file, dati), saranno intercettate dal service worker .

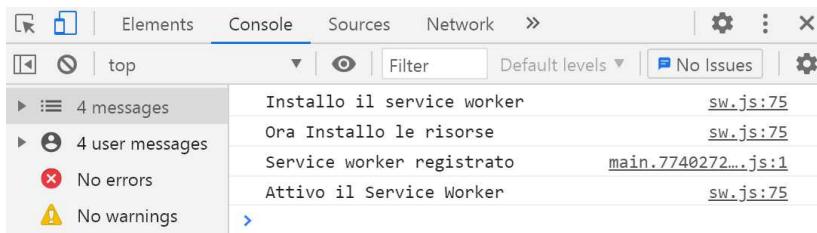


Figura 17.14 – Ciclo di vita di una PWA con service worker

Questo a patto d'inserire un opportuno gestore d'evento, che nel nostro caso è l'evento `fetch`.

```
sw.js
//...
self.addEventListener('fetch', e => e.respondWith(fetchRisorsa(e)));
async function fetchRisorsa(e) {
    console.log('Interetto richiesta ' + e.request.url);
    // recupero le risorse o dalla cache o dalla rete
}
```

Tutte le volte in cui l'applicazione viene chiusa o nei periodi di inattività dell'utente, il service worker viene messo in uno stato inattivo (*idle*) dal browser al fine di risparmiare risorse utili, soprattutto nei dispositivi mobili.

Se ora chiudiamo il browser e ci ricolleghiamo all'applicazione per simulare un secondo accesso, in teoria non dovremmo vedere né la fase d'installazione delle risorse, né quella legata all'attivazione. Dovremmo però vedere la fase di `fetch`, visto che ora tutte le richieste passano tramite il service worker.

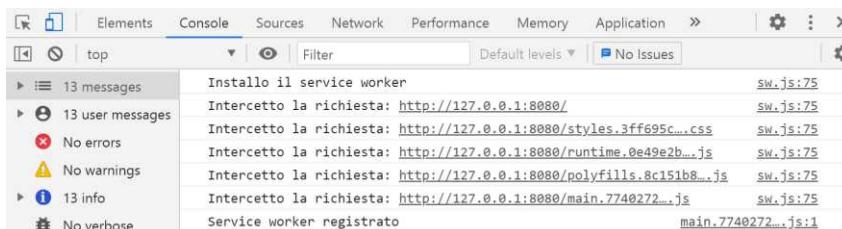


Figura 17.15 – Secondo accesso a una PWA con service worker già attivato

Concludendo, possiamo dire che, rispetto alle applicazioni Angular che sfruttano la libreria `@angular/pwa`, l'attivazione di una nuova versione, avviene subito in seguito al refresh della pagina, anche se l'applicazione è già aperta in più finestre.

Conclusione e Codice Personale

Siamo arrivati alla conclusione di queste 24 ore di corso e devo congratularmi con te per essere arrivato fino a qui.

Se partivi da zero, o quasi, non sarà stato un percorso agevole, ma spero di essere riuscito nel mio intento, ossia quello di farti capire le basi del funzionamento del framework Angular, nel più breve tempo possibile con un codice semplice ma funzionale all'obiettivo.

Ci sarebbero altri mille possibili approfondimenti, quindi ti suggerisco di seguirmi consultando il sito <https://www.video-corsi.com>

Ti invito a scrivere il tuo feedback a caldo su quello che hai imparato e a pubblicarlo nel caso sia positivo (**4 e 5 stelle** corrispondono ad un **feedback positivo**, le altre negativo), per incentivare altre persone a conoscere Angular e fargli capire che non è così difficile.

Nel caso appartenessi a quella piccolissima e fisiologica cerchia di finti o veri delusi, certo mi dispiace, ma se è una delusione sincera, allora **offrirti il mio aiuto** è il minimo che possa fare, quindi invece di chiuderti in te stesso con la pubblicazione di un commento che forse non ti serve a molto, approfittane per scrivermi personalmente all'indirizzo: assistenza9@video-corsi.com

Rileggi con calma le sezioni più “ostiche” e testa il tuo apprendimento sul campo, provando a riscrivere i diversi pezzi dell'applicazione sviluppata nel libro, provando anche a svolgere le diverse esercitazioni richieste.

Tutto il codice scritto nel libro lo puoi **scaricare gratuitamente**, registrandoti al link qui sotto. Dovrai inserire il tuo nome, l'indirizzo email che usi su Amazon e una password di almeno 5 caratteri.

<https://su.video-corsi.com/registrati.php>

NB: Tieni presente che saranno presenti più versioni del codice, e che probabilmente alcune righe saranno leggermente diverse da quelle presenti nel libro, in seguito ai frequenti aggiornamenti di Angular.

Per attivare il download del codice, entra nella sezione “Elenco Libri”, poi clicca sul bottone “codice” in corrispondenza al libro, e inserisci questo dato personale: **8GWF EKTGH23W94** (tutto in maiuscolo) o quello presente nell’ultima pagina del libro a fianco di “Printed in”.

Ti invito a segnalarmi privatamente eventuali errori di battitura o sul codice in modo che possa correggerli anche sul libro. Ogni nuovo aggiornamento introduce potenzialmente nuovi errori, quindi grazie fin da ora se vorrai scrivermi.

Buona programmazione!

Un abbraccio

Davide Copelli {ing.}

APPENDICE

I test per il recupero e l'invio delle informazioni dell'applicazione MetroChat, posso essere condotti sfruttando un'API realizzata per l'occasione e raggiungibile con i comandi indicati qui sotto. Per ogni chiamata, sono registrati l'indirizzo IP, la data e l'ora, in modo da bloccare eventuali richieste multiple che potrebbero rallentare il server. Ti invito ad usarla con moderazione.

L'indirizzo principale dell'API è stato modificato in : <https://www.dcopelli.it/angular/> anche se puoi continuare a usare l'indirizzo presente nel codice di ogni capitolo ossia <https://www.dcopelli.it/test/angular/>

Qui sotto, sono indicati la tipologia di chiamata (GET, POST) e il relativo percorso.

* **GET /treni/** - Restituisce la lista dei treni in arrivo presso un'ipotetica stazione.

Nella demo verranno restituiti tre treni, con un ritardo alla partenza variabile. Una volta che l'ultimo treno è partito, dovrai rieseguire l'app per avere nuovamente la lista dei treni aggiornata.

URL di chiamata: <https://www.dcopelli.it/angular/treni/>

Formato della risposta: oggetto JSON con singola chiave indicata qui sotto:

nome	tipo	Significato del valore
dati	Array	Il valore è un array costituito da oggetti JSON con la lista di chiavi evidenziata qui sotto.

Oggetto JSON (metro):

nome	tipo	Significato del valore
idt	string	Identificativo del treno
linea	string	Nome della linea
direzione	string	Ultima stazione in quella direzione
numchatting	string	Numero persone in chat in quel treno
tempo	number	Orario di partenza espresso in secondi dalla data di Unix.
stazione	string	Nome della stazione d'arrivo
carrozza	string	Nome della carrozza con più utenti in chat

* **POST /treni/** - Restituisce i dati di dettaglio del treno

Le intestazioni devono contenere l'identificativo del treno ***idt*** e l'API restituisce i dati del treno corrispondente.

URL di chiamata: <https://www.dcopelli.it/angular/treni/>

Formato della risposta: oggetto JSON con singola chiave indicata qui sotto:

nome	tipo	Significato del valore
dati	Array	Il valore di dati è un array costituito da un singolo oggetto JSON, rappresentativo del treno, con la seguente sequenza di chiavi.

Oggetto JSON (metro):

nome	tipo	Significato del valore
idt	string	Identificativo del treno
linea	string	Nome della linea
direzione	string	Ultima stazione in quella direzione
numchatting	string	Numero persone in chat in quel treno
tempo	number	Orario di partenza espresso in secondi dalla data di Unix.

* **GET /chat/** - Restituisce la lista dei messaggi chat di uno specifico treno

Nell'url devono essere passati come query i dati relativi all'identificativo del treno tramite ***idt***, l'identificativo dell'utente con ***idu=99*** (fisso nella demo) e il dato opzionale relativo alla stringa di ricerca ***search***. L'API restituisce la lista dei messaggi chat scambiati in quello specifico treno.

URL di chiamata: <https://www.dcopelli.it/angular/chat/?idt=XX&idu=99&search=YY>

Formato della risposta: oggetto JSON con singola chiave indicata qui sotto:

nome	tipo	Significato del valore
dati	Array	Ogni elemento dell'array è un oggetto JSON con la lista di chiavi evidenziata qui sotto.

Oggetto JSON (messaggio):

nome	tipo	Significato del valore
idm	string	Identificativo del messaggio
testo	string	Testo del messaggio

idu	string	Identificativo dell'utente
idt	string	Identificativo del treno
iddestinatario	number	Identificativo del destinatario del messaggio privato.

* **POST /chat/send/** - Permette di memorizzare un messaggio chat.

Le intestazioni devono contenere un oggetto JSON con chiavi **idm, idt, idu, testo** e l'API restituisce il messaggio completo comprensivo dell'identificativo aggiornato del messaggio.

URL di chiamata: <https://www.dcopelli.it/angular/chat/send/>

Formato della risposta: oggetto JSON con singola chiave indicata qui sotto:

nome	Tipo	Significato del valore
dati	Array	Ogni elemento dell'array è un oggetto JSON con la lista di chiavi evidenziata qui sotto.

Oggetto JSON (messaggio):

nome	tipo	Significato del valore
idm	string	Identificativo del messaggio
testo	string	Testo del messaggio
idu	string	Identificativo dell'utente
idt	string	Identificativo del treno
iddestinatario	number	Identificativo del destinatario del messaggio privato.

* **PUT /chat/send/preferiti/** - Abilita/Disabilita l'indicazione associata al simbolo di "preferiti" per un messaggio chat.

Le intestazioni devono contenere un oggetto JSON con chiavi **idutente, idmessaggio, stato** e l'API restituisce una stringa che indica la riuscita o meno dell'operazione.

URL di chiamata: <https://www.dcopelli.it/angular/chat/send/preferiti/>

Formato della risposta: stringa con valori "1" o "0"

* **GET /amici/** - Restituisce la lista degli amici per uno specifico utente (nella demo l'utente è sempre lo stesso)

Nell'url devono essere passati come query i dati relativi all'identificativo dell'utente tramite **idu**. L'API restituisce i dati dell'utente compreso un array con la lista di tutti gli amici.

URL di chiamata: <https://www.dcopelli.it/angular/amici/?idu=XX>

Formato della risposta: oggetto JSON indicato qui sotto:

nome	tipo	Significato del valore
id	string	Identificativo dell'utente
nome	string	Nome dell'utente
cognome	String	Cognome dell'utente
email	String	Email dell'utente
ip	String	Indirizzo IP dell'utente
amici	Array	Ogni elemento è un oggetto con chiavi id, nome e citta

SITOGRAFIA

<https://angular.io> – Guida Ufficiale Angular
<https://www.typescriptlang.org> – TypeScript Guida Ufficiale
<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide> - JavaScript Guida
<https://rxjs-dev.firebaseio.com> - RxJS Guida Ufficiale
<https://www.learnrxjs.io> - Elenco operatori RxJS con esempi
<https://ngrx.io> – Guida ufficiale NgRx
<https://mobx.js.org> – Libreria per la gestione dello stato
<https://developer.mozilla.org/en-US/docs/Web/API/>