# Angular: introduzione al framework, consigli, best practices

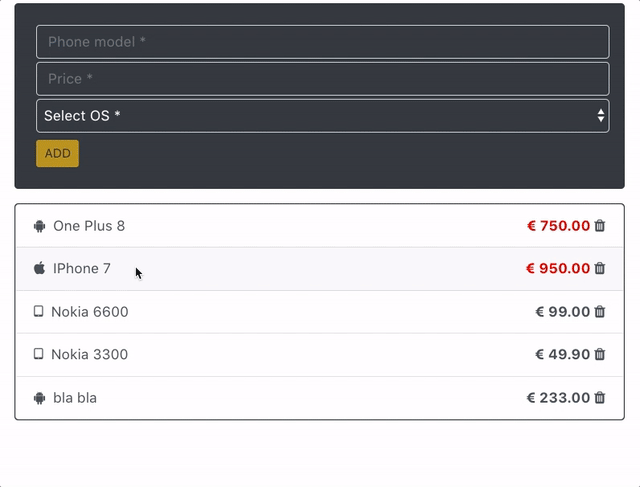
In questo articolo, descrivo alcune delle funzionalità principali incluse nella release 5 di Angular (UPDATE: l’articolo è stato aggiornato alla versione 6), uno dei framework più utilizzati nella creazione di Single-Page Applications (SPA).

*Con il termine****Single-page application****si intende un’applicazione web o un sito web che può essere usato o consultato su una singola pagina web con l’obiettivo di fornire una esperienza utente più fluida e simile alle applicazioni desktop dei sistemi operativi tradizionali. Wikipedia*

Gli argomenti principali trattati nell’articolo sono i seguenti:

* Creazione progetti Angular tramite l’utilizzo di angular-cli
* Utilizzo delle direttive incluse nel framework
* Dynamic Styling
* Template driven form e validators
* Creazione server mock API REST
* Comunicazione con il server tramite API REST

Il risultato finale dell’applicazione sarà un elenco di smartphone con un form per la gestione di tutte le operazioni CRUD:  
lettura (GET), inserimento (POST), cancellazione (DELETE) e modifica degli elementi (PUT/PATCH).



NodeJS e NPM

Per installare gli strumenti necessari a svolgere questo tutorial è indispensabile effettuare il download di [NodeJS](https://nodejs.org/en/" \t "_blank), installarlo e verificare la possibilità di utilizzare il comando npm (Node Package Manager) da terminale, che è incluso nell’installer di Node.

***TIP****: Un’alternativa all’installazione “tradizionale” di Node è l’utilizzo di****NVM****(Node Version Manager) grazie al quale è possibile utilizzare diverse versioni di Node nella stessa macchina. Molto utile se avete un progetto che richiede una versione precedente di Node rispetto a quella attuale o se volete creare ambienti separati in cui installare versioni differenti dei tool e/o provare diverse configurazioni. Disponibile per*[*Windows*](https://github.com/coreybutler/nvm-windows)*e*[*Mac*](https://github.com/creationix/nvm)*(anche tramite HomeBrew)*

Angular: creazione progetto

Il modo più semplice per configurare un progetto Angular è senza dubbio l’utilizzo di [angular-cli](https://github.com/angular/angular-cli" \t "_blank), un’altro strumento utilizzabile da riga di comando.

1. npm install -g @angular/cli

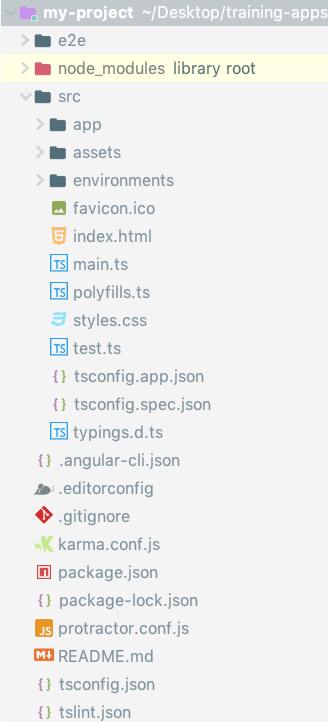
Dopo aver atteso qualche secondo / minuto sarà quindi possibile generare un progetto Angular utilizzando semplicemente il modulo ng, disponibile dopo l’installazione della CLI (Command Line Interface).

*In Windows è spesso necessario chiudere il terminale e riaprirlo affinché possiate utilizzare gli strumenti installati. Nel caso non fossero disponibili verificare di avere i permessi di amministrazione del sistema dato che questi strumenti modificando le variabili di ambiente. In caso negativo, provate ad aprire il terminale da amministratore e installate nuovamente il pacchetto*

1. ng new my-project

*L’operazione richiederà qualche minuto a seconda della velocità della propria connessione internet*

Il risultato sarà la creazione di una struttura simile alla seguente (potrebbe variare leggermente in base alla versione di angular-cli che state utilizzando):



Una volta terminato il processo, il progetto potrà essere avviato utilizzando il comando npm start

1. cd my-project
2. npm start #oppure ng serve

Aprendo il browser e visitando l’url http://localhost:4200 sarà visibile un’applicazione demo con un semplice “hello world”.

Installare Bootstrap e Font-Awesome

Installiamo ora [Bootstrap 4](https://getbootstrap.com/) (framework CSS) e [Font-Awesome](http://fontawesome.io/) (set di icone) allo scopo di migliorare il look&feel della nostra demo con estrema facilità.  
Ci tengo a precisare che utilizzeremo solo il CSS di Bootstrap e non i componenti Javascript/jQuery forniti da Bootstrap. Esiste infatti una versione di Bootstrap creata appositamente per Angular ma questo argomento non sarà trattato in questo articolo.

*Naturalmente potrete utilizzare qualunque altro framework CSS o creare il proprio CSS da zero*

Installiamo quindi i due pacchetti utilizzando npm:

1. npm install bootstrap@4.0.0 font-awesome@4.7.0 --save

*L’opzione --save non è più necessaria dalla versione 5 di npm perché abilitata di default*

O la più concisa:

1. npm i bootstrap@4.0.0 font-awesome@4.7.0

*Ho specificato esplicitamente la versione delle librerie da utilizzare (@x.x.x) per far in modo che il tutorial sia utilizzabile anche in futuro, quando saranno disponibili nuove versioni delle librerie.*

Apriamo ora il file angular-cli.json, individuiamo il nodo styles e aggiungiamo i path ai css delle due librerie appena installate in modo tale da renderle accessibili globalmente.

|  |  |
| --- | --- |
| 1  2  3  4  5 | "styles": [    "../node\_modules/bootstrap/dist/css/bootstrap.min.css",    "../node\_modules/font-awesome/css/font-awesome.min.css",    "styles.css"  ], |

*UPDATE: In angular 6, il nome del file di configurazione della CLI è cambiato in angular.json e il path alla cartella node\_modules non necessità più del ../ ma sarà semplicemente node\_modules/bootstrap/ecc*

Per far in modo che la modifica abbia effetto è necessario “killare” il processo npm start avviato in precedenza (utilizzando ad esempio CTRL + C o CMD + C) e riavviarlo con npm start.

***TIP****: Angular-cli utilizza WebPack dietro le quinte, un mix tra un automation tool e un module bundler, allo scopo di compilare il progetto, generare le build, gestire CSS/SASS e molto altro. Il file angular-cli.json permette di configurare alcune funzionalità di WebPack senza tuttavia dover necessariamente conoscere lo strumento. Maggiori informazioni sul*[*repository GitHub di angular-cli*](https://github.com/angular/angular-cli)

Creare un server Mock per le API REST

Esistono diversi strumenti in grado di creare un set di servizi (mock) REST in pochi minuti ma uno dei più apprezzati è senza dubbio [json-server](https://github.com/typicode/json-server" \t "_blank).

Il funzionamento è molto semplice. Si crea un file .json, si avvia *json-server* da riga di comando e in pochi secondi saranno disponibili delle API REST per la manipolazione del JSON:

* GET: per recuperare le informazioni dal JSON
* POST: per inserire contenuti
* PUT / PATCH: per aggiornare parzialmente o in toto un elemento
* DELETE: per rimuovere un elemento

Sarà quindi possibile installare *json-server* globalmente utilizzando la riga di comando:

1. npm install -g json-server

oppure come dipendenza di un eventuale progetto che state sviluppando:

1. npm install --save-dev json-server

***TIP****: utilizzo -save-dev affinché il pacchetto sia disponibile tra le devDependencies e non sia quindi inserito nel bundle finale del progetto (ovvero la versione da distribuire)*

Dopo aver installato il pacchetto sarà sufficiente creare un file, ad esempio db.json, e inserire all’interno di questo file una struttura JSON. Per il nostro esempio creeremo una lista di devices:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25 | {    "devices": [      {        "label": "One Plus 8",        "os": "android",        "price": 750,        "rate": 3,        "memory": 7400,        "desc": "One Plus 5 is a flagship phone...",        "id": 2      },      {        "label": "IPhone 7 +",        "os": "ios",        "price": 700,        "rate": 3,        "memory": 3000,        "desc": "",        "id": 24      }    ],    "login": {      "token": "efoiwejo32r32-fake-token"    }  } |

Per avviare il server sarà quindi sufficiente posizionarsi da terminale nella cartella in cui si è creato il file db.json e avviare digitare la seguente istruzione:

1. json-server --watch db.json

Ora avremo a disposizione un set di API per manipolare la nostra collezione di devices.

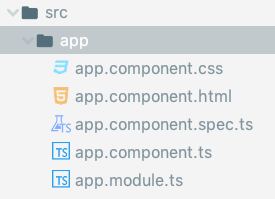
Sarà infatti possibile effettuare operazioni in POST, GET, PUT, PATCH e DELETE semplicemente invocando l’endpoint generato dal server, in questo caso:

1. http://localhost:3000/devices
2. http://localhost:3000/login

Angular: comunicazione con il server utilizzando HttpClient

Carichiamo ora una collezione dati dal server mock creato con json-server.

La cartella /src contiene il file app.component.ts che rappresenterà il nostro entry-point, ovvero il componente principale della nostra applicazione.



Apriamo il file e modifichiamo il contenuto come segue:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | // app.component.ts  import { Component } from '@angular/core';  import { HttpClient } from '@angular/common/http';    @Component({    selector: 'app-root',    templateUrl: './app.component.html',    styleUrls: ['./app.component.css']  })  export class AppComponent {      constructor(private http: HttpClient) {      this.getAll();    }      getAll() {      this.http.get('<http://localhost:3000/devices>')        .subscribe(result => console.log(result));    }  } |

A questo punto avremo due problemi:

1) Se non avete avviate il server JSON, come descritto in precedenza, non avrete a disposizione l’end-point dal quale recuperare la collezione dati.  
Il mio consiglio è quello di creare una cartella server nella root del progetto (quindi fuori dalla cartella /src) e di avviare il comando da terminale : json-server --watch server/db.json.

***TIP****: è possibile aggiungere un azione al nodo scripts all’interno del file package.json per automatizzare il processo a avviarlo semplicemente con l’istruzione: npm run server  
"server": "json-server --watch server/db.json"*

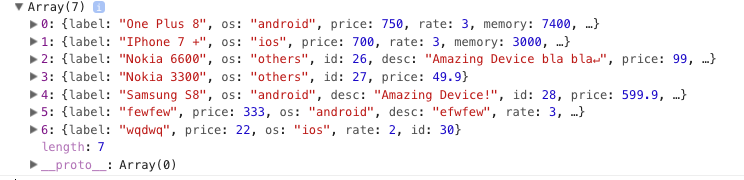
2) Il secondo problema è rappresentato dall’utilizzo di HttpClient, un servizio incluso in Angular per la comunicazione con il server, senza aver tuttavia importato il relativo modulo.

***TIP****: Angular è suddiviso in differenti moduli, ovvero delle collezioni di componenti, direttive, servizi (e molto altro) “pre-confezionati” che potete utilizzare nelle vostre applicazioni: FormsModule per i form, HttpClient per la comunicazione con il server, BrowserModule per le direttive base e così via*

E’ quindi necessario aprire il file src/app.module e importare HttpClientModule. Visto che ci siamo, importiamo anche FormsModule che ci servirà in seguito per la gestione dei form 😉

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | // app.module.ts  import { BrowserModule } from '@angular/platform-browser';  import { NgModule } from '@angular/core';    import { AppComponent } from './app.component';  import { HttpClientModule } from '@angular/common/http';  import { FormsModule } from '@angular/forms';    @NgModule({    declarations: [      AppComponent    ],    imports: [      BrowserModule, HttpClientModule, FormsModule    ],    providers: [],    bootstrap: [AppComponent]  })  export class AppModule { } |

Salviamo tutti i files, avviate il browser e navigate all’indirizzo http://localhost:4200. Aprite i Dev Tools (se usate Chrome: F12 su windows oppure CMD + OPTIONS + i su Mac) e dovreste vedere il risultato nella console dei dev tools:



Model e tipizzazione

Per sfruttare le potenzialità del linguaggio Typescript, alla base del framework Angular, possiamo creare un custom type che rappresenti i nostri device.  
Sarà utile per tutta una serie di motivi: abilitare intellisense e autocompletamento negli editor/IDE, ricevere errori dal compilatore qualora utilizzassimo delle proprietà errate, documentare il codice, solo per citarne alcuni.

Creiamo il file device.ts in una nuova cartella src/model/

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | // src/model/device.ts  export interface Device {    id?: number;    label?: string;    os?: string;    price?: number;    memory?: number;    rate?: number;    desc?: string;  } |

***TIP****: il punto di domanda (?) indica che la proprietà sarà facoltativa. Sarebbe preferibile evitarlo in un contesto reale ma, ai fini didattici, se ipotizziamo di non gestire tutte le proprietà di un device, come in questo scenario, è preferibile utilizzare questo approccio per evitare errori del compiler.*

Modifichiamo ora il file app.component.ts allo scopo di:

1) utilizzare il nuovo tipo Device  
2) salvare l’intera collezione dei dati ricevuti dal server in un array di Device

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | // src/app.component.ts  import { Component } from '@angular/core';  import { HttpClient } from '@angular/common/http';  import { Device } from './model/device';    @Component({    selector: 'app-root',    templateUrl: './app.component.html',    styleUrls: ['./app.component.css']  })  export class AppComponent {    devices: Device[];      constructor(private http: HttpClient) {      this.getAll();    }      getAll() {      this.http.get<Device[]>('<http://localhost:3000/devices>')        .subscribe(result => this.devices = result);    }  } |

*Tramite l’istruzione this.http.get< Device[] > specifichiamo che il risultato ottenuto dal server sarà di tipo Device: una sorta di casting del risultato. Non è utile in questo scenario ma in contesti reali è pressoché indispensabile se vogliamo evitare errori del compilatore qualora provassimo ad accedere al contenuto di una proprietà utilizzando ad esempio l’istruzione devices[0].anyProperty.*

Template: visualizzare una collezione di dati

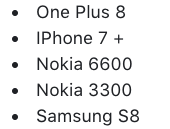
Per visualizzare un elenco di elementi, possiamo utilizzare la direttiva ngFor fornita da Angular (modulo BrowserModule).  
Questa direttiva permette di effettuare un ciclo su una collezione dati e di renderizzare ogni elemento tramite un template HTML.

***TIP****: le direttive sono dei “componenti speciali” che possono essere applicati al DOM o ad altri componenti. Angular include un set di direttive molto utili per manipolare il dom (ngIf, ngClass, ecc.) ma potete chiaramente creare direttive personalizzate*

Apriamo quindi il file src/app.component.html, cancelliamo il contenuto hello world e inseriamo il seguente codice:

|  |  |
| --- | --- |
| 1 | <li \*ngFor="let device of devices">{{device.label}}</li> |

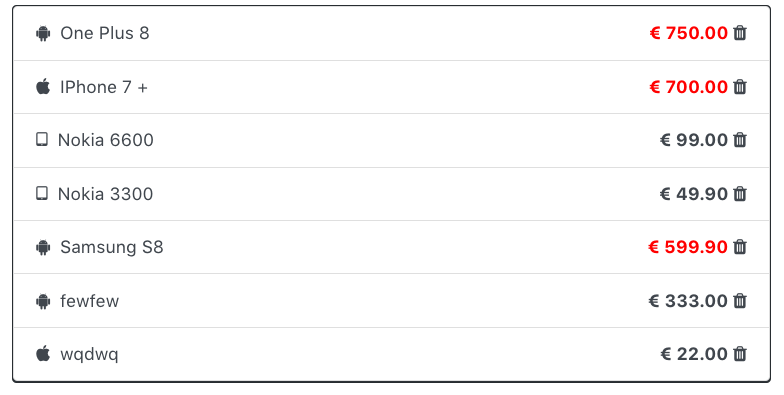
che produrrà il seguente risultato:



***TIP****: si utilizzano le parentesi graffe, come in {{device.label}}, per eseguire e processare istruzioni Javascript all’interno del template HTML. In questo caso visualizziamo la label di un singolo device. Nel caso inserissimo al loro interno, ad esempio, un’operazione matematica, ad es. {{1+1}}, avremmo come output il risultato: 2*

Possiamo ora completare il template visualizzando altre proprietà del device e utilizzando alcune delle classi CSS messe a disposizione da Bootstrap e FontAwesome.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35 | <div class="card bg-dark text-white mb-3">      <!--Devices List -->    <div \*ngFor="let device of devices"         class="list-group-item list-group-item-action">        <!--os icon-->      <i class="fa"         [ngClass]="{              'fa-android': device.os === 'android',              'fa-apple'  : device.os === 'ios',              'fa-tablet' : device.os === 'others'           }"      ></i>          <!--label-->      <i class="fa fa-in"></i>      <span>{{device?.label}}</span>        <!--display rate-->      <!--...missing...-->        <div class="pull-right">        <!--price -->        <strong \*ngIf="device.price"                [style.color]="device.price > 500 ? 'red' : null">          € {{device.price | number: '1.2-2'}}        </strong>          <!--trash icon-->        <i class="fa fa-trash icon"></i>      </div>    </div>  </div> |



Cancellazione e selezione di un elemento

Innanzitutto aggiungiamo i metodi setActive e delete al nostro componente.  
Il primo sarà invocato al click di ogni elemento della lista, allo scopo di selezionarlo.  
Il secondo, invece, al click dell’icona “trash”, il cestino.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41 | // src/app.component.ts  import { Component } from '@angular/core';  import { HttpClient } from '@angular/common/http';  import { Device } from './model/device';    @Component({    selector: 'app-root',    templateUrl: './app.component.html',    styleUrls: ['./app.component.css']  })  export class AppComponent {    devices: Device[];    active: Device = {};      constructor(private http: HttpClient) {      // console.log ('environment:', env);      this.getAll();    }      getAll() {      this.http.get<Device[]>('<http://localhost:3000/devices>')        .subscribe(result => this.devices = result);    }      setActive(device: Device) {      this.active = device;    }        delete(event: MouseEvent, device: Device) {      event.stopPropagation()      this.http.delete<any>(`http://localhost:3000/devices/${device.id}`)        .subscribe(          () => {            const index = this.devices.indexOf(device)            this.devices.splice(index, 1);          }        );    }    } |

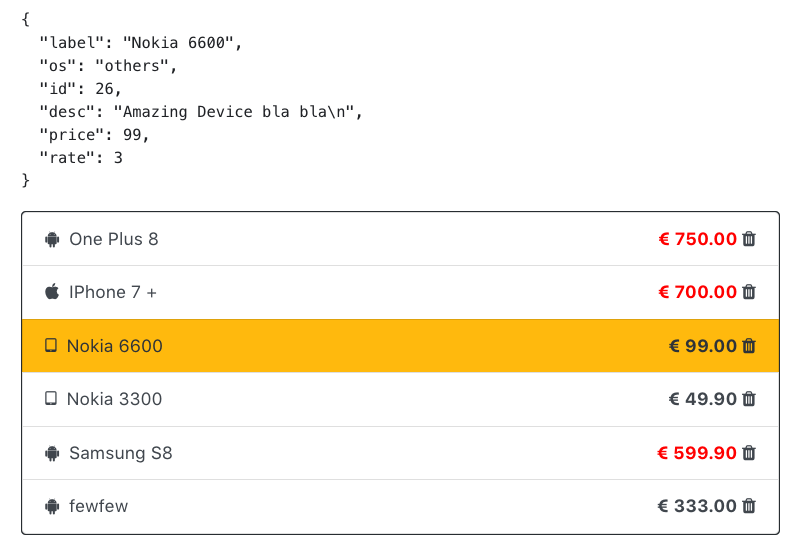
*Il metodo setActive salva una reference dell’elemento in una proprietà active che sarà successivamente utilizzata per popolare il form ed effettuare la modifica delle proprietà del device selezionato*

Modifichiamo quindi il template per invocare i due metodi appena creati e per visualizzare l’elemento selezionato proprio sopra la lista (in cui successivamente posizioneremo il form).  
Utilizziamo inoltre la direttiva ngClass per evidenziare l’elemento selezionato.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40 | <pre>{{active | json}}</pre>    <div class="card bg-dark text-white mb-3">      <!--Devices List -->    <div \*ngFor="let device of devices"         class="list-group-item list-group-item-action"         [ngClass]="{'bg-warning text-dark': device.id === active?.id}"         (click)="setActive(device)">        <!--os icon-->      <i class="fa"         [ngClass]="{              'fa-android': device.os === 'android',              'fa-apple'  : device.os === 'ios',              'fa-tablet' : device.os === 'others'           }"      ></i>          <!--label-->      <i class="fa fa-in"></i>      <span>{{device?.label}}</span>        <!--display rate-->      <!--...missing...-->        <div class="pull-right">        <!--price -->        <strong \*ngIf="device.price"                [style.color]="device.price > 500 ? 'red' : null">          € {{device.price | number: '1.2-2'}}        </strong>          <!--trash icon-->        <i class="fa fa-trash icon"           (click)="delete($event, device)"></i>      </div>    </div>  </div> |

***TIP****: l’istruzione json utilizzata in {{active | json}} permette di visualizzare il contenuto di una collezione dati all’interno di un template ed è chiamata “pipe” (ex filter in AngularJS)*

Il risultato sarà il seguente:



*Come potete notare notato che al metodo delete passo la proprietà $event, ovvero l’evento del mouse generato dal click dell’utente. In questo modo è possibile invocare l’istruzione event.stopPropagation() per evitare che anche il metodo setActive sia invocato al click sull’icona “trash”. Maggiori info sulla*[*documentazione MDN*](https://developer.mozilla.org/en-US/docs/Web/API/Event/stopPropagation)

Angular Form

Una delle funzionalità più amate di Angular è la gestione dei form, superiore per quantità di feature e potenzialità a qualunque altra libreria o framework attualmente disponibile sul mercato (e cito ad es. React o Vue).

Angular fornisce due soluzioni per la gestione dei form:

* Template driven form: utilizzati in questo articolo
* Reactive form: un approccio che sfrutta il paradigma della programmazione funzionale, davvero molto potente e flessibile (utilizzando RxJS, incluso come dipendenza del framework)

L’argomento “Form” è davvero molto ampio e in questo articolo analizzeremo velocemente solo una piccola parte delle funzionalità dei template-driven form.

Iniziamo creando un semplice form proprio sopra la lista:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31 | <div class="card bg-dark text-white mb-3">    <!--edit / add form-->    <form novalidate          (submit)="save(f)"          #f="ngForm"          class="card-body">        <input type="text"             [ngModel]="active?.label"             name="label"             required             class="form-control bg-dark text-white"             placeholder="Phone model \*"             >        <div class="btn-group btn-group-sm m-1">        <button class="btn btn-warning"                type="submit"                [disabled]="f.invalid">          {{active?.id ? 'SAVE' : 'ADD'}}        </button>        <button            class="btn btn-light"            type="button"            \*ngIf="active?.id"            (click)="reset()">          ADD NEW        </button>      </div>    </form>  </div> |

Salvando il file e provando il form riceveremo degli errori perché non abbiamo ancora implementato i metodi save() e reset().

Il form non è ancora completo ma di seguito descrivo alcune delle funzionalità utilizzate finora:

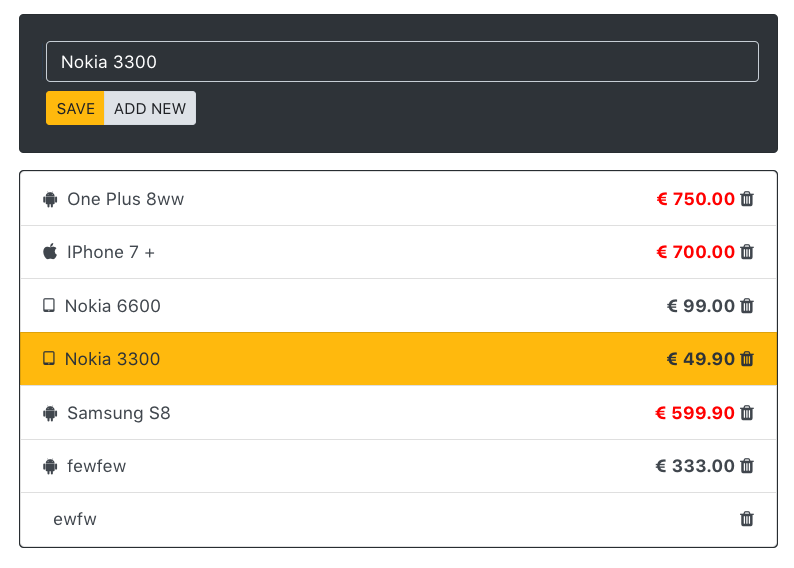
* + Il form sarà utilizzato sia per gestire l’inserimento di nuovi elementi, che la modifica di quelli esistenti
  + Al submit del form (tramite click o pressione del pulsante INVIO della tastiera) sarà invocato il metodo submit(f) tramite il quale passiamo anche una reference del form

*L’istruzione #f viene definita “template reference variable” e rappresenta una reference al nostro form. Tramite questa proprietà potremo conoscere in ogni momento lo stato del form: se è valido (f.valid o f.invalid), se è già stato utilizzato / sporcato (f.dirty), possiamo recuperare le informazioni inserite in tutti i campi del form tramite la proprietà “value” (f.value) e molto altro. Lo stesso meccanismo può essere utilizzato anche sul singolo campo. Si potrà quindi sapere se un campo è valid, invalid, dirty, touched, errors e molto altro. Davvero potente!*

* + La direttiva ngModel permettere di sincronizzare una proprietà della classe con il campo di input tramite l’utilizzo di un binding 1-way (al contrario di AngularJS che utilizza il binding bidirezionale). Sostanzialmente, quando il valore della proprietà active muta, anche il campo di input sarà aggiornato con il relativo valore. Quindi ogni qualvolta l’utente selezionerà un elemento della lista, il contenuto della proprietà active.label sarà visualizzata all’interno del campo di input.

***TIP****: è possibile abilitare il binding bidirezionale utilizzando le doppie parentesi [(ngModel)] ma è preferibile utilizzare un approccio 1-way. I motivi sono molteplici ma, per citarne uno, l’applicazione risulterà più semplice da mantenere evitando di perdere il controllo del flusso dati. In questo esempio potrebbe sembrare inutile ma nel momento in cui l’applicazione dovrà scalare, la UI sarà suddivisa in centinaia di componenti, si gestirà la business logic in servizi o si utilizzeranno pattern architetturali come [Redux](https://redux.js.org/" \t "_blank) o [Mobx State Tree](https://github.com/mobxjs/mobx-state-tree" \t "_blank), sarà di fondamentale importanza. In questo tutorial non affronteremo nessuno di questi argomenti ma vi consiglio di abituarvi a lavorare nel modo corretto fin da subito*

* Il pulsante per il submit sarà disabilito fino a che tutti i campi di input del form non saranno validi. In questo esempio abbiamo specificato che la label è un elemento required perciò il form non sarà valido fino a che quel campo (ed eventuali altri) non sarà valido
* La *label* del pulsante *submit* assumerà il valore SAVE, nel momento in cui abbiamo un elemento selezionato, oppure ADD quando invece ne inseriremo uno nuovo
* Il pulsante *reset* invece si occuperà di deselezionare l’elemento attivo, per permettere la creazione di un nuovo elemento



Modifica e Inserimento elementi

Completiamo la demo aggiungendo i metodi per la gestione dell’inserimento e modifica di elementi e integrando, nel template HTML, i campi di input e una select per la gestione di alcune delle proprietà dei device (prezzo e sistema operativo)

app.component.js

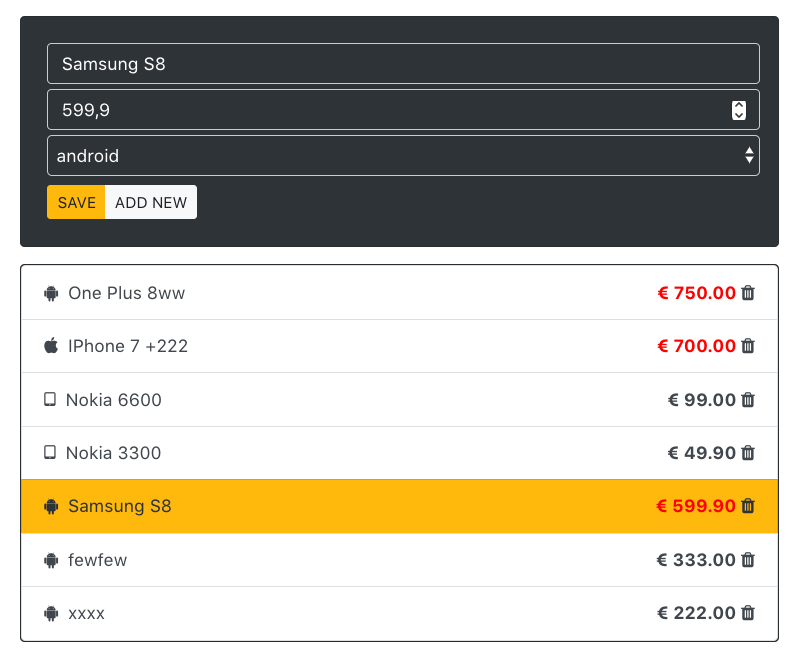
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81 | import { Component } from '@angular/core';  import { HttpClient } from '@angular/common/http';  import { Device } from './model/device';  import { NgForm } from '@angular/forms';    const INITIAL\_STATE = { label: null, os: null };    @Component({    selector: 'app-root',    templateUrl: './app.component.html',    styleUrls: ['./app.component.css']  })  export class AppComponent {    devices: Device[];    active: Device = INITIAL\_STATE;      constructor(private http: HttpClient) {      // console.log ('environment:', env);      this.getAll();    }      getAll() {      this.http.get<Device[]>('<http://localhost:3000/devices>')        .subscribe(result => this.devices = result);    }      setActive(device: Device) {      console.log( device )      this.active = device;    }        delete(event: MouseEvent, device: Device) {      this.http.delete<any>(`http://localhost:3000/devices/${device.id}`)        .subscribe(          () => {            const index = this.devices.indexOf(device)            this.devices.splice(index, 1);          }        );    }      save(form: NgForm) {      if (this.active.id) {        this.edit(form.value);      } else {        this.add(form.value);        form.reset();      }    }      add(device: Device) {      this.http.post<Device>(`http://localhost:3000/devices`, device)        .subscribe(res => {          this.devices.push(res)          this.reset();        })    }      edit(device: Device) {      const newDevice = Object.assign(        {},        this.active,        device      );        this.http.patch<Device>(`http://localhost:3000/devices/${newDevice.id}`, newDevice )        .subscribe(          res => {            const index = this.devices.findIndex(device => device.id === newDevice.id) ;            this.devices[index] = newDevice;          }        );      }      reset() {      this.active = INITIAL\_STATE;    }    } |

app.component.html

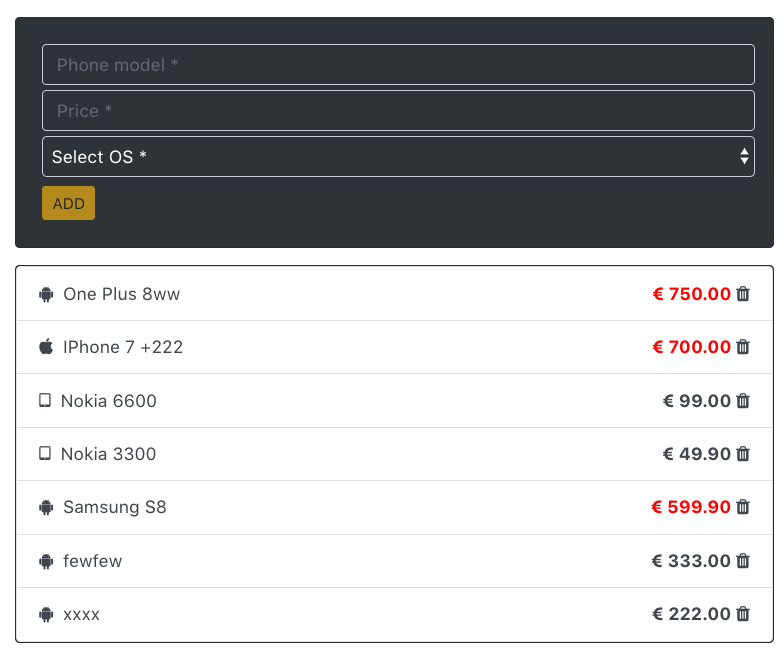
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89 | <div class="card bg-dark text-white mb-3">    <!--edit / add form-->    <form novalidate          (submit)="save(f)"          #f="ngForm"          class="card-body">        <input type="text"             [ngModel]="active?.label"             name="label"             required             class="form-control bg-dark text-white m-1"             placeholder="Phone model \*">          <input type="number"             [ngModel]="active?.price"             name="price"             class="form-control bg-dark text-white m-1"             placeholder="Price \*">        <select [ngModel]="active?.os"              name="os"              required              class="form-control bg-dark text-white m-1">        <option value="null">Select OS \*</option>        <option value="ios">ios</option>        <option value="android">android</option>        <option value="others">others</option>      </select>          <div class="btn-group btn-group-sm m-1">        <button class="btn btn-warning"                type="submit"                [disabled]="f.invalid">          {{active?.id ? 'SAVE' : 'ADD'}}        </button>        <button            class="btn btn-light"            type="button"            \*ngIf="active?.id"            (click)="reset()">          ADD NEW        </button>        </div>    </form>  </div>      <div class="card bg-dark text-white mb-3">      <!--Devices List -->    <div \*ngFor="let device of devices"         class="list-group-item list-group-item-action"         [ngClass]="{'bg-warning text-dark': device.id === active?.id}"         (click)="setActive(device)">        <!--os icon-->      <i class="fa"         [ngClass]="{              'fa-android': device.os === 'android',              'fa-apple'  : device.os === 'ios',              'fa-tablet' : device.os === 'others'           }"      ></i>          <!--label-->      <i class="fa fa-in"></i>      <span>{{device?.label}}</span>        <!--display rate-->      <!--...missing...-->        <div class="pull-right">        <!--price -->        <strong \*ngIf="device.price"                [style.color]="device.price > 500 ? 'red' : null">          € {{device.price | number: '1.2-2'}}        </strong>          <!--trash icon-->        <i class="fa fa-trash icon"           (click)="delete($event, device)"></i>      </div>    </div>  </div> |

Risultato:

In fase di editing:



In fase di inserimento:



Next Step

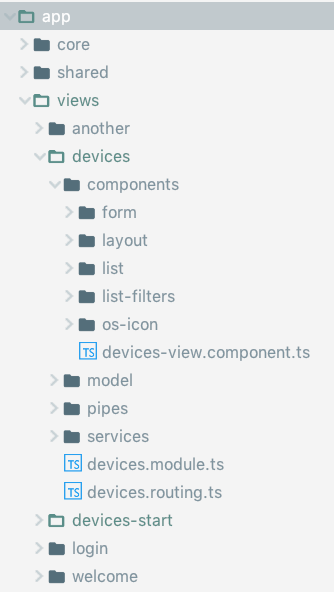
Questo articolo introduce solo una minima parte delle funzionalità del framework e non vengono applicate tutta una serie di best practices e metodologie indispensabili per rendere l’applicazione più scalabile, manutenibile e testabile.  
Il framework include, infattim moltissimi altri strumenti tra i quali:

* La possibilità di suddividere il codice in componenti e creare quindi custom HTML tags (ad es. )
* Suddividere la business logic in servizi utilizzando il motore di dependency injection integrato
* Organizzare il progetto in diverse route e in moduli custom
* Creare custom directives e custom pipes
* Gestire autenticazione (ad es. JWT o OAuth), integrare interceptor ecc.
* Utilizzare facilmente pattern architetturali per la gestione dello stato applicativo come Redux o MobxState Tree, già citati in precedenza
* e molto molto altro

Ad esempio, l’esercizio precedente, in un contesto reale, potrebbe essere scritto come segue, e questo è solo uno dei possibili approcci che non utilizza neppure uno dei pattern architetturali menzionati sopra:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38 | @Component({    selector: 'devices-view',    template: `            <toggable [title]="Device Form">            <add-edit-form                [active]="store.active"                (reset)="actions.reset()"                (save)="actions.save($event)">            </add-edit-form>          </toggable>            <toggable title="DEVICES" [closable]="false">              <list-filter                class="header-content"                [filters]="filters"                (update)="this.filters = $event"></list-filter>              <devices-list                [devices]="store.devices"                [active]="store.active"                [filters]="filters"                (selectRow)="setActive($event)"                (delete)="actions.delete($event)"></devices-list>          </toggable>    `  })  export class DevicesViewComponent {    constructor(      public store: DeviceStore,      public actions: DeviceService,    ) {      // Load devices      this.actions.getAll();    }    } |

Lo stesso discorso vale per la struttura del progetto che, al crescere della complessità, naturalmente dovrà essere organizzato diversamente. Un esempio:



DEMO

Di seguito una demo live. Tieni presente che il server Node utilizzato è un hosting gratuito. Potrebbe non essere velocissimo (soprattutto in fase di avvio) e non mi ritengo responsabile dei contenuti inseriti dagli altri utenti 😉

[Visualizza lo script su StackBlitz](https://stackblitz.com/edit/angular5-bootstrap4-crud-device-list-simple?embed=1&file=app/views/devices/devices.component.ts)  
(nella demo ho gestito anche il “rate” del device, la descrizione e viene utilizzato inoltre Angular Router per la creazione di applicazioni multi-view)