Come installare Angular

1)Per installarlo, eseguire il seguente comando:

npm install -g @angular/cli

2)Per creare un nuovo progetto Angular con Angular-cli, basta posizionarsi con il terminale o prompt dei comandi sulla cartella che abbiamo scelto per ospitare il nostro codice, e digitare:

ng new my-app //dove my-app sta per il nome dell'app

3)Rispondere **yes** all'angular routing ed accettare CSS come formato per lo stile.

4)Per avviare l'app eseguire nel terminale integrato:

ng serve

NB: il terminale integrato lo si apre facendo tasto destro sulla cartella del progetto e cercando terminale integrato.

Creazione di un componente

ng g c nome_componente

g sta per generate

c sta per component

Il nuovo componente verrà aggiunto in automatico in app.module.ts.

Ogni componente creato risulta il figlio di: app.components.ts

Anatomia di un Client Angular

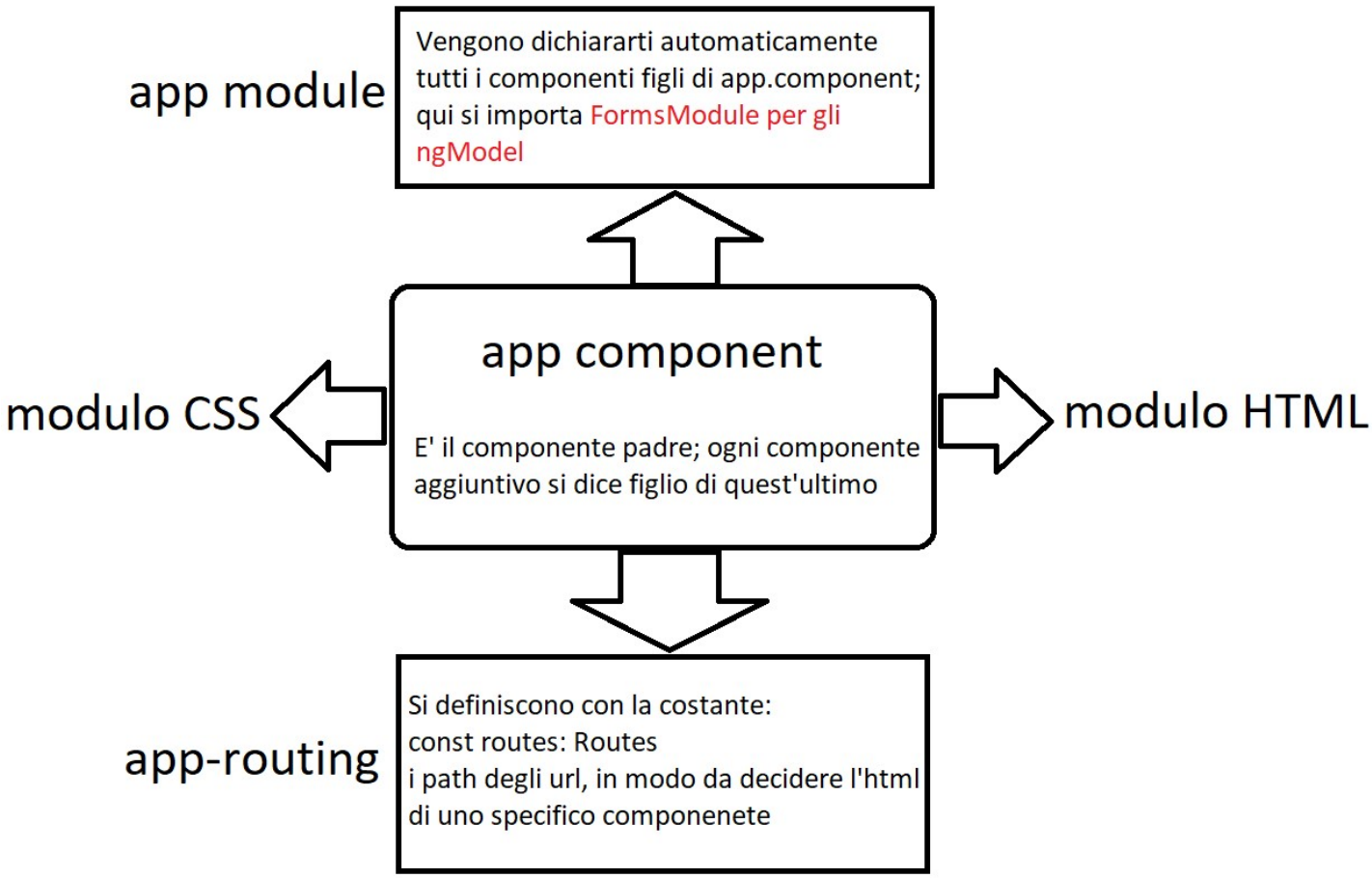
Un programma Client nasce con lo scopo di mandare richieste (trasmissione di dati) ad un programma Server che sulla base dei dati ricevuti invia una risposta offrendo un servizio. Tuttavia, può lavorare da solo per piccole operazioni.

L'anatomia di un Client Angular vede come nucleo (il cuore del programma) una cartella **src/app** con all'interno il componente app. Quest'ultimo è il componente padre; tutti gli altri componenti che verranno aggiunti saranno automaticamente i figli di app.component.

Il componente padre (app.component) rispetto ai figli prevede:

-app.module.ts;

-app.routing;



Come i figli deve invece prevedere un file .ts che lo dichiara, un file .css ed un file.html.

Il Client Angular lavora su un un'unica pagina HTML.

Il client che andiamo a sviluppare possiede un'unica view, ovvero, prevede una sola pagina web dinamica che corrisponde ad app.component.html. Questa pagina fa da finestra all'html dei vari componenti figli. Ammettiamo di metterci in app.component.html; vediamo la seguente sequenza di tag:

```
<app-navbar></app-navbar>
<!--router-outlet mostra l'html dei componenti in cui path sono
stati definiti nella costante routes in app-routing.module.ts-->
<router-outlet></router-outlet>
<app-footer></app-footer>
```

L'unica view del Client è composta quindi dall'html del componente navbar.component, dall'html di footer.component e dall'html del componente proiettato dal tag **router-outlet**. Quest'ultimo proietta l'html di un componente attraverso un path specifico. Per chiarire il concetto bisogna spostarsi in app-routing.module.ts. Nella costante routes, indichiamo con quale path il tag router-outlet deve mostrare l'html di specifici componenti.

```
const routes: Routes = [
  { path : 'add', component: FormUtenteComponent },
  { path : 'list', component: ListUtentiComponent },
  { path: '', redirectTo: 'add', pathMatch: "full"}
];
```

In riferimento all'immagine di cui sopra, l'url: http://localhost:4200/add

vedrà la view mostrare:

- l'html del componente Navbar;
- l'html del componente FormUtente
- l'html del componente Footer

In corrispondenza dell'url: http://localhost:4200/list la view mostrerà:

- l'html del componente Navbar;
- l'html del componente ListUtenti
- l'html del componente Footer

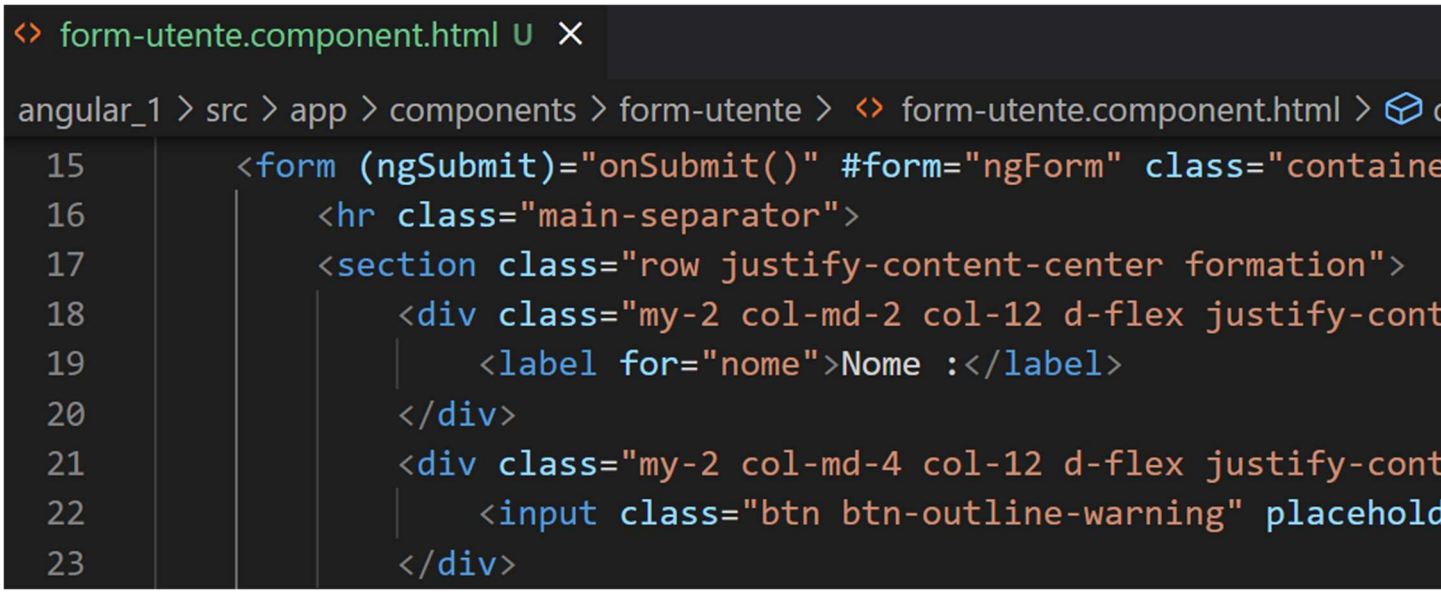
Quindi col tag router-outlet abbiamo una finestra dinamica, mentre i tag: app-navbar ed app-footer sono finestre statiche.

Il terzo path che usa la proprietà redirectTo indica che di default la view mostrerà l'html di FormUtente in corrispondenza della finestra router-outlet. Attenti a non dimenticare patchMatch settato a full, altrimenti la proiezione non avverrà.

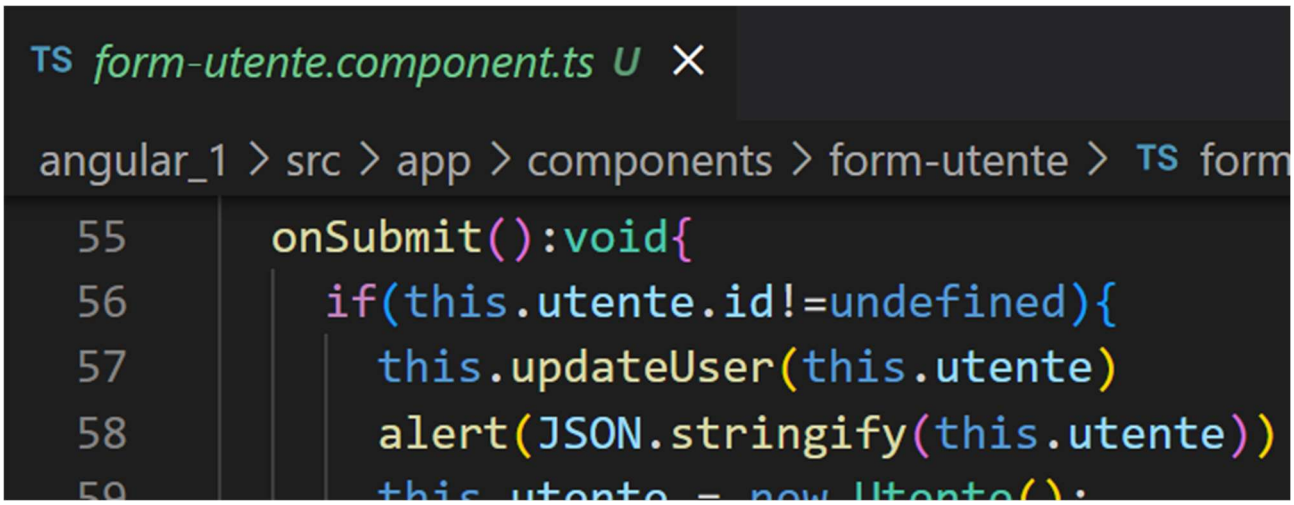
Le classi che implementano l'interfaccia OnInit

Mettendo da parte app.component.ts su cui non metteremo mano, nei file: **nome_componente_figlio.component.ts** troveremo una classe che implementa l'interfaccia ***OnInit*** da cui ereditare il metodo ngOnInit che inizializza in memoria il componente. In questa classe, i metodi dichiarati, saranno associati al solo componente figlio di riferimento e non potranno essere usati dagli altri. Dobbiamo immaginare il file.ts, il file.css ed il file.html come costituenti un unico componente in quanto collegati tra loro; per tanto, solo dall'html del componente figlio si potrà accedere ai metodi dichiarati nel suo file.ts. Si dice che i metodi del file.ts sono **accoppiati** al componente figlio specifico.

Prendiamo in esame il form di form-utente.component.html mostrato in parte dalla seguente immagine:



Ci aspettiamo che la chiamata del metodo onSubmit() esegua l'algoritmo del metodo dichiarato in form-utente.component.ts:



Il file html è strettamente collegato al file ts dello stesso componente figlio; un secondo componente non può avere quindi accesso ad onSubmit() oltre che form-utente.component.ts.

Recupero dati dai form

1)@ViewChild

@ViewChild è il decoratore utilizzato per recuperare i valori degli elementi della view ed assegnarli agli attributi della classe implementate OnInit(). Per fare un esempio, partiamo da un form:

viewchildtable.component.html U ✕ TS viewchildtable.component.ts U

mytable > src > app > components > viewchildtable > viewchildtable.compo

```
1   <form class="container form mt-4">
2   <p>{{titolo}}</p>
3   <br>
4   <input class="btn btn-outline-warning" #rif_nome plac
5   <br>
6   <input class="btn btn-outline-warning" #rif_cognome p
7   <br>
8   <input class="btn btn-outline-warning" #rif_eta place
```

Col cancelletto #rif-elemento indichiamo univocamente uno specifico elemento; il valore di quest'ultimo passerà agli attributi di tipo ElementRef annotati col decoratore @ViewChild:

```
export class ViewchildtableComponent implements OnInit {

  titolo = "Inserisci persona: ";
  @ViewChild('rif_nome') tagNome!: ElementRef;
  @ViewChild('rif_cognome') tagCognome!: ElementRef;
  @ViewChild('rif_eta') tagEta!: ElementRef;
```

Ogni elemento contrassegnato col # deve prevedere un (input) associato al metodo che eseguirà il passaggio di valori dall'html al ts:

```
#rif_nome placeholder="Nome: " (input)="inputValori()"><br>

#rif_cognome placeholder="Cognome: " (input)="inputValori()">

#rif_eta placeholder="Età: " (input)="inputValori()"><br>
```

I valori inseriti nei relativi input vengono immediatamente passati al metodo inputValori() del file ts. Ogni attributo annotato con @ViewChild recupera il valore dal form attraverso la proprietà nativeElement.value

```
nome: string = "";
cognome: string = "";
eta: number = 0;

constructor() { }

ngOnInit(): void {
}

inputValori() {
  this.nome = this.tagNome.nativeElement.value;
  this.cognome = this.tagCognome.nativeElement.value;
  this.eta = this.tagEta.nativeElement.value;
}
```

I valori vengono successivamente passati agli attributi: nome, cognome, eta. Quest'ultimi, possono essere usati per dare i valori in output nell'html da dove provenivano. L'html dovrà avere le expression language {{nome_attributo}} che mostrano l'output dell'attributo indicato.

```
<table class="table table-striped">
<tr><th width="150px">Nome</th><th wid
<tr>
    <td>{{nome}}</td>
    <td>{{cognome}}</td>
    <td>{{eta}}</td></tr>
</table>
```

Questo scambio classe-view viene detto **Binding Bidirezionale**. Dalla view recuperiamo i dati per darli alla classe, dopo di che col metodo della classe li rimandiamo alla view. Il risultato finale dell'esempio è il seguente:

2)Passaggio come argomento del metodo

E’ possibile passare i valori degli elementi indicati dal # come argomenti di un metodo chiamato con un evento “click”.

```
<div class="my-2 col-md-4 col-12 d-flex justify-conten
    <input class="btn btn-outline-danger" #nome placeh
</div>
<div class="my-2 col-md-2 col-12 d-flex justify-conten
    <!--nome.value recupera il valore dall'input const
    <button (click)="submitNewUser(nome.value)" class=
</div>
```

L’argomento deve essere passato come: rif#.value

Il metodo nel file ts riceverà il valore come segue:

```
submitNewUser(nome:string):void{
  alert(nome+" è stato aggiunto alla lista!");
  this.lista.push(nome);
  alert("La tabella si è aggiornata!");
  alert("Tuttavia, automaticamente verrà resett
}
```

NB: i dati ricevuti dal .value arrivano sempre come stringa, quindi qualora li si volesse come numeri, utilizzare le funzione parseInt() o parseFloat() per ottenere numeri interi o decimali.

Ammettiamo di voler sviluppare una calcolatrice, dove ogni operazione aritmetica corrisponde ad un bottone che con un click richiama una funzione che vuole due numeri come argomenti. Nella view vediamo:

```
<form class="container form mt-4">
    <br><br>
    <input class="form-control btn-outline-warning" #numero1
    <br><br>
    <input class="form-control btn-outline-warning" #numero2
    <br><br>
```

```
<section class="row justify-content-center formation">
<button class="btn btn-primary" (click)="somma(numero1.value, numero2.value)"
<button class="btn btn-secondary" (click)="sottrazione(numero1.value, numero2.
<button class="btn btn-danger" (click)="prodotto(numero1.value, numero2.value)
<button class="btn btn-warning" (click)="divisione(numero1.value, numero2.valu
```

Nel ts:

```
somma(numero1:string, numero2:string):void{
  var n1:number = parseInt(numero1);
  var n2:number = parseInt(numero2);
  this.opService.servSomma(n1, n2);
}

sottrazione(numero1:string, numero2:string):void{
  var n1:number = parseInt(numero1);
```

3)Recupero dati da un form template driven

Il form prende le sembianze di un oggetto di tipo model, per cui gli input sono strettamente legati agli attributi della classe entity.

Partiamo dal seguente model:

```
export class Utente{
    nome:string = "";
    cognome:string = "";
    eta:number = 0;
}
```

Il form template driven si presenta come segue:

```
<form (ngSubmit)="onSubmit()" #form="ngForm" cla
    <hr class="main-separator">
    <section class="row justify-content-center f
        <div class="my-2 col-md-2 col-12 d-flex
            <label for="nome">Nome :</label>
        </div>
        <div class="my-2 col-md-4 col-12 d-flex
            <input [(ngModel)]="utente.nome"  cl
        </div>
    </section>
    <section class="row justify-content-center f
        <div class="my-2 col-md-2 col-12 d-flex
            <label for="cognome">Cognome :</labe
        </div>
        <div class="my-2 col-md-4 col-12 d-flex
            <input [(ngModel)]="utente.cognome"
        </div>
```

Il form dovrà avere la direttiva #form=”ngForm” ed i moduli [(ngModel)] a cui si associano i valori con la coppia oggetto.campo. Il submit del form crea un oggetto di tipo model (Utente in tal caso), facendo new+costruttore, e lo assegna ad un attributo della classe nel ts del componente figlio. Ammettiamo di dichiarare il seguente attributo:

utente: Utente = new Utente();

```
export class FormUtenteComponent implements OnInit {

  utente:Utente = new Utente();
```

Con i 3 [(ngModel)] di cui le seguenti assegnazioni:

“utente.nome”, “utente.cognome”, “utente.eta”

Si indicherà che all’attributo utente:Utente viene passato un oggetto con all’interno i valori dei 3 campi (nome, cognome, eta) presi dal form. L’attributo utente:Utente è quindi direttamente collegato all’ngModel del form html.

Per abilitare i moduli (ngForm) ed (ngModel) occorre importare la classe **FormsModule** in app.modules.ts.

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

@NgModule({
  declarations: [
    AppComponent,
  ],
  imports: [
    BrowserModule,
    AppRoutingModule,
    //FormsModule è importante per usate ngModel ed ngForm
    FormsModule,
```

Il Dependency Injection di Angular

Finora abbiamo visto che per ogni componente di Angular si ha una coppia file.ts-file.html dove i metodi del ts sono connessi alla pagina html; da quest'ultima è infatti possibile attivare i metodi con eventi come: (ngSubmit) e (click). Tuttavia, è possibile creare una classe di servizio disaccoppiata da uno specifico file html, con l'obiettivo di rendere i suoi metodi condivisibili da tutti i componenti di Angular. Il **disaccoppiamento** prende il nome di **Dependency Injection (DI)** in quanto l'oggetto servizio, per essere usato da uno dei componenti, deve essere ***iniettato*** nel costruttore di quest'ultimo. La classe servizio, per essere iniettabile, deve contenere l'annotazione: @Injectable.

```
@Injectable({
  providedIn: 'root'
})
export class UtenteCrudService {
```

dove **providedIn: root** significa che il servizio sarà disponibile a livello di applicazione come singleton, ovvero, che la classe ammetterà un unico oggetto messo a disposizione per l'intero programma. A seguire, un esempio di come iniettare l'oggetto di tipo UtenteCrudService nel costruttore di un componente (es. FormUtente):

```
export class FormUtenteComponent implements OnInit {

  //Nel costruttore inietto l'oggetto con cui richiamare
  //i metodi servizio (in questo caso utente-crud.service.ts)
  constructor(private utenteService:UtenteCrudService) { }
```

Asincronia delle comunicazioni

Nello sviluppo di applicazioni frontend, qualsiasi sia il framework utilizzato (Angular, Vue.js, Jquery, etc..), ci si scontra ben presto con il concetto di asincronia delle comunicazioni.
E' infatti comune che all'interno della nostra app vengano fatte delle chiamate a dei servizi remoti per richiedere dei dati, chiamate di cui non si conosce il tempo di risposta: per questo motivo non si possono gestire queste richieste in maniera sincrona (ovvero, eseguendo riga per riga rispettando una fissata sequenza), dato che, questo porterebbe ad un "freeze" dell'interfaccia per tutto il tempo di attesa della risposta.
Ecco che quindi ci vengono in aiuto gli **Observable**, oggetti che riescono a gestire in maniera efficiente le operazioni asincrone.
Observable rappresenta il core type di Reactive X, una libreria atta alla programmazione asincrona e "event-based". Si basa sul concetto che i dati futuri vengono "osservati", ovvero, tenuti sotto controllo dagli **Observer**, ovvero gli oggetti dei componenti che agiscono come "consumers" e che come tali si registrano su quei dati (diventando appunto "**subscribers**") e quindi possono gestirli non appena questi saranno disponibili. Questa registrazione avviene tramite il metodo subscribe() di Observable, il quale accetta per l'appunto il dato richiesto come argomento.
Procediamo con un esempio:
ammettiamo di avere come Observer la classe ListUtentiComponent che sottoscrive il recupero di una lista di oggetti di tipo (model) Utente (argomento del subscribe). Il metodo con sui fa la sottoscrizione (subscribe) viene chiamato da un oggetto Observable ritornato da un metodo della classe di servizio iniettabile UtenteCrudService. Partiamo dalla classe ListUtentiComponent.

```
export class ListUtentiComponent implements OnInit {

  //Questa lista aggiorna automaticamente in output la tabella
  //ListUtentiComponent
  utenti:Utente[] = [];

  constructor(private utenteService:UtenteCrudService) { }

  getUsers():void{
    this.utenteService.getUsers().subscribe((data:Utente[])=>{
      console.log("Utenti presi!")
      this.utenti = data;
    })
  }
```

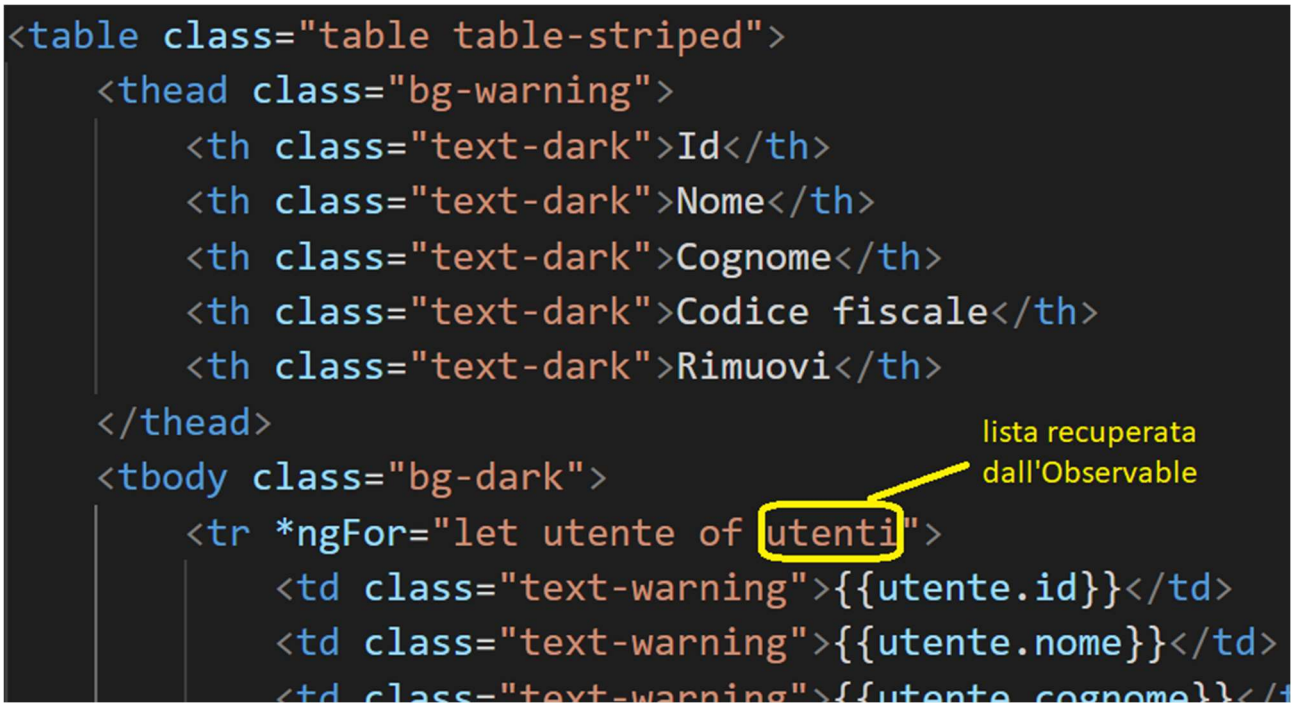
this.utenteService.getUsers() chiama il metodo omonimo (nell'immagine sottostante) del servizio iniettabile UtenteCrudService:

```
getUsers():Observable<Utente[]>{
  return this.http.get<Utente[]>(`${this.uri}/crud`)
}
```

Come mostrato, il metodo ritorna l'oggetto Observable<Utente[]> con cui chiamare il metodo subscribe.

Con http.get si manda una richiesta di tipo get al server (ad esempio un gestionale sviluppato con il framework Express JS) dove l'uri rappresenta l'url con cui raggiungere il server stesso. La risposta del server si tradurrà con il ritorno dell'oggetto Observable<Utente[]> che contiene la lista richiesta. Questa viene poi passata al subscribe che assegna la lista recuperata all'attributo utenti:Utente[] della classe ListUtentiComponent;

essendo quest'ultimo collegato ad un file html sarà possibile visualizzare la lista tramite browser.




```
<table class="table table-striped">
    <thead class="bg-warning">
        <th class="text-dark">Id</th>
        <th class="text-dark">Nome</th>
        <th class="text-dark">Cognome</th>
        <th class="text-dark">Codice fiscale</th>
        <th class="text-dark">Rimuovi</th>
    </thead>
    <tbody class="bg-dark">
        <tr *ngFor="let utente of utenti">
            <td class="text-warning">{{utente.id}}</td>
            <td class="text-warning">{{utente.nome}}</td>
```