Convenzioni: dopo il nome dell’ng module.component tutto piccolo. Nome della classe in grande.

Es. user.component.ts

user.component.html

user.component.css

user.component.spect.ts 🡪 serve per i test in fase di sviluppo. Può non essere presente

Nome classe: UserComponent

@Component 🡪 Decoretor è una funzione quindi si mettono le parentesi tonde e poi le graffe perché mettiamo degli oggetti dentro. Selector, templateUrl perché indichiamo il file HTML che andiamo ad implementare, styles: [] in styles andiamo ad inserire il css.

{{}} 🡪Le doppie graffe nel file HTML indicano che si può prendere il contenuto dell’oggetto dichiarato nel file ts. Qualunque oggetto.

Per ogni ngModule conviene creare una cartella perché avere un ordine dei module.

{{ array | json}} 🡪 |(pipe) 🡪 serve per poter scorrere un array e stampare i dati in json

\*ngFor=”let user of users” in html serve per la costruzione delle tabelle e vuol dire per ogni utente dell’array utenti fai il ciclo for.

Il component service serve per separare la logica dai dati. Quando si collega con il server comunica con esso per ricevere i dati.

const 🡪 se una variabile non cambierà mai

let 🡪 se la variabile è una variabile momentanea

Angular injection:

mettere all’interno del costruttore le injection nel seguente modo:

Invece di mettere new nel costruttore inserire la classe da utilizzare come nel seguente esempio:

construct(serviceUser: UserService){} e viene iniettato come servizio. Per fare questo UserService deve essere inserito in app.module.ts e deve essere dichiarato dentro provider [] in modo

@Injectable 🡪 è un decoratore che si usa in Service per indicare che il servizio può avere delle dipendenze e prima di creare un sigleton se è già presente viene utilizzata sempre la stessa istanza.

ngOnInit 🡪 è un metodo dell’interfaccia OnIniti e, quindi, deve essere implementato. Ogni volta che un metodo viene compilato poi non devono essere implementate le interfacce anche se è una buona abitudine implementarle in typescript. Esempio:

export class UpdateDistintaComponent implements OnInit

In ngOnInit si possono implementare i componenti di ts invece che nel construct.

Nel construct:

constructor(public activeModal: NgbActiveModal, private distintaApi: DistintaApiService,

private usersApi: UsersApiService, private rowApi: RowApiService,

private modalService: NgbModal) {

Tutto questo fa parte dell’injection

Modo per creare un component da riga di comando:

* ng generate component nomeComponent
* ng g c nomeComponent

Per mettere un componente in html in un altro component si usa il comando

<app-User></app-User> in html 🡪visualizza il component in un altro component.

Esempio: <app-commission-details></app-commission-details>

Dove app-commission-details è il nome dichiarato nel selector details e viene poi inserito in Html del component app-commission-update nell’html.

Questo serve per creare una sorta di componente padre (commission-update) e componente figlio (commission-details).

Come si fa a passare una variabile da componente padre a componente figlio?

Nel componente padre nell’html passiamo la variabile nel seguente modo:

<app-User [user]=”user”></app-User> 🡪 dove [user] è il nome della variabile che vogliamo passare; “user” è il valore della variabile.

Nel componente figlio dobbiamo dichiarare la variabile dicendogli che la variabile è una variabile di tipo input. Per dire in angular che è una variabile di tipo input, nel componente figlio la dichiariamo nel seguente modo:

@Input() user 🡪 quindi usiamo un decoratore di input che indica che si possono ricevere dei dati in questa variabile

All’interno delle parentesi tonde possiamo usare un alias in modo che se cambia qualcosa all’interno all’esterno poi non cambia nulla

Esempio:

@Input(‘data\_user’) user

Abbiamo dato ad user l’alias data\_user

Un altro modo per dichiarare le variabili è di inserirle all’interno del @Component dove c’è il selector scrivendo:

inputs[

‘nome-Input: alias’

], ……

Una o l’altra forma va bene.

Se la variabile è composta da una serie di attributi che devono essere visualizzati in una tabella e distanziati nel selector invece di scrivere

Selector:’app-user’

Possiamo scrivere:

selector: :’tr [app-user]’

e nell’Html invece di scrivere:

<app-User [user]=”user”></app-User>

Mettiamo il valore direttamente nel tr nel seguente modo:

<tr app-user \*ngFor=”let user of users” [user-data]=”user”>

Le parentesi quadre servono per indicare che la variabile è una variabile in input.

Come intercettare gli eventi con Angular come, ad esempio, intercettare il click e mapparlo ad un metodo della classe di un componente

<button type="button" title="Cancella" class="btn btn-danger" (click)="parent.delete(account.id)">

(click) serve per indicare in angular un evento che viene gestito attraverso "parent.delete(account.id)"

Le donde in (click) stanno ad indicare che l’evento è un valore in uscita che chiama il metodo “parent.delete.

Click è l’evento nativo che si vuole chiamare; parent.delete è il metodo ed account.id sono le variabilli che si vogliono passare al metodo. Qiesto metodo è il modo in cui Angular intercetta gli eventi nativi sia del DOM che del Browser.

Metodo .splice(index, numero elementi da eliminare) per eliminare un element di un array.

Vediamo ora come passare valori da figlio a padre.

@Output(‘onDeleteUser’) userDeleted 🡪 tra parentesi abbiamo il nome pubblico e fuori il nome privato come nello stesso modo di input. Questo valore si dichiara nel component figlio.

@Output(‘onDeleteUser’) userDeleted= new EventEmitter(); 🡪 E’ un metodo di angular core (quindi si importa angular/core)

Nel metodo deleteUser() del componente figlio passiamo poi l’evento con il comando:

this.userDeleted.emit(this.user);

In questo modo noi abbiamo scatenato l’evento. Non lo stiamo ancora ascoltando. Per ascoltarlo ci si sposta nell’elemento padre nell’html e si aggiunge

<tr app-user \*ngFor=”let user of users” [user-data]=”user” (onDeleteUser)=”onDeleteUser($event)”>

E nel ts della classe padre dichiariamo il metodo nel seguente modo:

onDeleteUser(user){

//alert(user.lastname);

this.userService.delete(user)

}

Funziona come prima chiamando l’event nell’elemento padre.

INTERFACCE

In angular si possono creare delle Interface

Si possono creare da riga di comando con il comando:

* ng generate interface nomeInterfaccia

export interface User{

valore1: string;

valore2: numeber;

………

valoreN: object;

}

Ci serve per indicare e garantire che gli oggetti di quel tipo abbiano tutti gli stessi valori.

Per indicare, poi, ad esempio un array di tipo user si possono utilizzare due sintassi:

Array<User>

User[] = [{

},

{

}]

FORM:

\*ngIF.

Può mostrare o meno un elemento del DOM a seconda di una condizione booleana. In particolare mette l’elemento se la condizione è true altrimenti non lo mette proprio. E’ meglio rimuoverlo dal DOM se non è utilizzato per rendere più leggero il programma.

La condizione ngIf deve essere dinamica, ovvero deve verificarsi al verificarsi di un evento. Ad esempio se mettiamo un button per la modifica vogliamo che il form sia visibile solo nel caso della modifica. Ora vogliamo che cliccando sul tasto deve comparire il component dichiarato nell’\*ngIf

HTML5 i type degli input nell’html possono essere number, text, email

Di norma quando si passano eventi sempre con il decorator Output ed il metodo emitter. Da notare che l’emitter può essere anche castomizzato con uno specifico tipo.

@Output(‘onDeleteUser’) userDeleted= new EventEmitter<User>();

nell’html del padre:

<app-user (updateUser) =”updateUser($event)”></app-user>

<div class=”nomeClasse” \*ngIf=”userSelected”> 🡪$event=userSelected

Nel component principale

COME LEGARE UN FORM AD UN OGGETTO

Facendo il binding. Si può fare con la direttiva NgModel.

Per prima cosa per poter utilizzare ngModule si deve importare il Forms/module gi angular/forms, per avere accesso a tutta la parte di form e poi ngModule.

[(ngModule)] =la proprietà e presente sia in input che in output e se io modifico la proprietà in input modificherà anche il valore originale.

[ngModule]= la variabile è in input; 🡪 in questo caso se modifichiamo la variabile in input il valore nel form non cambia perché abbiamo un One way Databinding. Stiamo collegando solo in un senso ovvero dal componente al template e non dal form verso il componente.

(ngModule)= la variabile è in output;

Un oggetto in javascript viene passato per riferimento. Questo vuol dire che se cambio l’oggetto dentro la funzione che riceve quell’oggetto cambia anche l’oggetto originale. Per evitare che cambi l’oggetto originale quello che si può fare è passare una copia dell’oggetto.

Per fare una copia di un oggetto un metodo è quello di utilizzare

const oggettoCopia= Object.assign({},{nomEoggettoDaCopiare}) 🡪{} è l’oggetto target in questo esempio un oggetto vuoto.

Per passare un campo nascosto nell’html si usa:

<input type=”hidden” name=”id” [(ngModule)]=”user.id”> 🡪 lo passiamo nascosto e poi lo possiamo usare, ad esempio, per verificare se quell’id è vuoto o per salvare un nuovo utente.

Codice che ci ritorna l’index di un array.

const idx= this.users.findIndex((v) => v.id === user.id);

if(idx !== -1){

this.users[idx]=users;

}

This.users.spice(0,0,user) 🡪 In questo caso non cancello ma dico che alla posizione 0 voglio inserire un nuovo user.

In un metodo nel ts se facciamo un form in html possiamo passare la variabile form. Per fare questo possiamo usare una variabile che si chiama template variable e per averla in html scriviamo:

<form class=nome-classe #f=”ngForm” > 🡪 il # indica che si tratta di una variabile template di nome f. In questo caso poi noi possiamo accedere alla variabile del form chiamandola senza il cancelletto ad esempio:

{{f}}

{{f.form | json}} \_\_> possiamo vedere delle caratteristiche del form

{{form.form.value | json}}

🡪Per avere accesso ai valori del form

Quindi un altro modo di procedere invece di dichiarare la variabile input nel ts e poi di utilizzarla può essere quella di utilizzare la variabile form passandola nei metodi e poi prendere i dati di quella variabile. Si può procede in più modi.

Per passarlo dall’html nel metodo dichiarato in click passiamo (click)=nomeMetodo(f.form)

Nel ts:

nomeMetodo(f){}

f.form.reset() 🡪 per resettare i valori della form

COME CREARE LA COPIA DI UNA VARIABILE

Ad esempio creare la copia di una variabile input:

@Input user

Nel ts prima del costruttore facciamo in questo modo:

private userCopy: User;

private \_\_user:User;

e poi utilizziamo i metodi getter e setter in questo modo sempre prima del costructor:

@Input set user(user:User){

this.\_\_user=user;

this.userCopy=Object.assign({},user);

}

get user(){

return this.\_\_user;

}

Possiamo utilizzare questa cosa in un metodo reset dove diciamo che quando facciamo il reset la variabile è uguale alla vecchia copia nel seguente modo:

this.user=this.userCopy; 🡪 dove user è il nome della function getter e setter. Questo è un buon metodo per il reset delle form.

Per crare un nuovo progetto angular nel cli:

ng new NomeProgetto

PIPES

Dalla documentazione di Angular emerge che Pipe è una classe con un decorator @pipe che trasforma un valore in ingresso ritornando un valore. Angular possiede una bella lista di pipe già predefiniti. Vediamo alcuni esempi:

{{‘HIDRAN’ | lowecase}}

Nell’html a schermo restituirà :

hidran in minuscolo

Se, ad esempio, in ts abbiamo una variabile

Title= “nome mio” e la vogliamo passare nell’html in modo che si veda in maiuscolo scriviamo

<h1> {{ title | uppercase }}</h1>

<h1> {{ title | uppercase |titlecase }}</h1>

Io posso mettere più pipe per una variabile. Ad esempio in questo caso viene prima tutto trasformato in maiuscolo (uppercase) e poi solo la prima lettera di ogni parola che forma il title viene scritta in maiuscolo.

{{45.50 | currency}}

Mi restituirà: $45.50

{{45.50 | currency:’EUR’}}

Mi visuaizza l’euro prima del valore

{{45.50 | currency:’EUR’:’code’}}

Visualizza il codice, quindi

EUR 45.50

{{45.50 | currency:’EUR’:’code’: ‘2.2-4’}}

Per l’ultima parte, local si devono importare due componenti che si trovano nella documentazione di Angular e che devono essere importate in app-module.

Poi scriviamo:

RegisterLocaleData(localIt) ; subito dopo gli import inseriti

Fare il pipe di una data new Date()

{{ birthDay | date:’short/medium’ }}

Guardare la documentazione per tutti i formati di pipe.

Come creare una pipe personalizzata. Dal cli scivere

ng g pipe nomePipe

e poi modifichiamo la classe secondo le nostre esigenze.

.map 🡪 serve per mappare le modifiche ad ogni elemento di un array

Return nome.map() mi ritorna la funzione con i valori modificati

.join(‘’) serve per unire tutti i pezzi di un array in una stringa e per togliere le virgole

Quando finiamo di realizzare la pipe usiamo il decoratore pipe e diamo il nome alla pipe, come si fa per i component ma all’interno della funzione l’attributo è name: ‘nomePipe’

BOOTSTRAP VERSIONE 4

Bootstrap può essere installato con npm install bootstrap….. --save : c’è la riga da eseguire per l’installazione sul sito ufficiale di bootstrap

Una volta installato per includerlo nella cartella di lavoro:

andare su angular-cli.json – ed in “styles” aggiungere il seguente codice:

“../node\_modules/bootstrap/dist/bootstrap.min.css”,

class=”container-fluid” 🡪 è il contenitore principale dell’app e, quindi nell’html dovrebbe essere la prima riga.

Class=”row” è la classe di una riga. All’interno possiamo mettere

Class=”col-8” 🡪 Una riga può essere divisa in 12 colonne e ad ogni colonna diciamo quando spazio dare.

Class=”col-md-4” 🡪 stiamo creando una pagina responsive dove quando diminuiamo la dimensione la colonna va a capo

Class=”table 🡪 formatto come tabella table-striped 🡪 do le righe con colori alternati alla tabella table-sm” 🡪la dimensione della tabella deve essere small

Table-dark 🡪la tabelle viene tutta scura

class =btn btn-primary btn-sm 🡪per indicare un bottone piccolo

class =btn btn-danger

class= “form-group”

Ogni input di una form avrà una classe form-control

Per usare delle icone si usa in cli

Npm install –save font-awesome angular-font-awesome

Una volta istallato importiamo il modulo

{AngularFontAwesomeModule} nell’app-module

E và dichiarato in imports: dove c’è anche BrowserModule e va aggiunta la riga come per bootstrap su angular-cli.json – ed in “styles” aggiungere il seguente codice:

“../node\_modules/font-awesome/css/font-awesome.min”,

poi in html <fa name=”nomeIcona” size=”lg>

Le icone le possiamo utilizzare per i pulsanti

In <button><fa name=”pencil” size=”lg></button>

Bootstrap dipende da JQuery per tutta la parte di javascript quindi in angular-cli.json in scripts andrebbe inclusa tutta la parte di jquery e di bootstrap. Però quando utilizziamo angular dobbiamo evitare deglli scripts che modifichino il DOM direttamente. Quello che si può fare, quindi, è installare un package che si chiama ngBootstrap che con delle direttive fa queste cose per noi. Quindi troviamo il comando per installare il package. Dopo di che lo importiamo nell’app-module in imports scrivendo:

NgbModule.forRoot(),

ROOTING E NAVIGAZIONE

Con angular noi sviluppiamo una single page application che vuol dire applicazione a pagina singola nel senso che una volta che l’applicazione si carica da lì in poi non viene più richiesto nulla al server che non siano chiamate AJAX, ecc. ma la pagina non viene più ricaricata.

Import {RouterModule, Routes} from angular/core

Const routes : Routes = [

{

path: ‘users’,

component: UsersComponent

},

{

path:’’,

redirectTo:’users,

pathMatch: ‘full’

},

{

path: ‘users/new’,

component: UserDetailComponent

},

{

path: ‘users/:id/edit’,

component: UserDetailComponent

}

]

All’interno andiamo a scrivere tutte le rotte ed avremo un array di oggetti con tutte le rotte. Questo dopo l’import in app-module. Dopo di che in imports dobbiamo definire il modulo per le rotte e passare l’array nel seguente modo:

RouterModule.forRoot(routes); 🡪 forRoot indica le rotte principali, un altro metodo è forChild

Una volta fatto questo in HTML per specificare il percorso

<a class=”nav-link” href=”#” routerLink=”/”>Home</a>

Router Outlet

Se vogliamo prendere il valore :id di una rotta possiamo iniettare le costrittore il servizio ActivetedRoutecon il seguente codice:

private route: ActivatedRoute 🡪 noi possiamo abbonarci a questo servizio mettendo nell’ngOninit la seguente riga di codice:

this.route.params.subscribe(

(params) => {

This.user = this.userService.getUser(+params.id);

}

)

subscribe serve per indicare che ci stiamo abbonando a questo servizio. All’interno di subscribe creimao una arrow function al cui interno avremo una variabile params

per fare il cast a number basta mettere:

+params.id

Per passare la rotta da un component nel costruttore scriviamo

Private route: Router

E poi usiamo il servizio route nel seguente modo:

this.route.navigate([ ‘users’,this.user.id,’edit]);

Questo corrisponde a costruire la rotta:

‘users/:id/edit’

N.B 🡪 Anche in angular quando si ragiona con l’indice di un array si deve considerare che è indice-1

Se in un if, ad esempio metto per tre volte uguale vuol dire che deve essere dello stesso tipo e dello stesso valore (===)

This.router.navigate lo possiamo usare, ad esempio quando dopo un save vogliamo vedere una specifica schermata.

COMUNICAZIONE TRA SERVER E CLIENT FRONTEND

Laravel è un framework che può essere usato per implemtare il beckend

.env: in questo file va sistemata la parte di comunicazione con il beckend.

Return Responce()->json(message) 🡪 per indicare che inviamo un messaggio in formato json

C’è la parte request e la parte responce tra server e client

HTTPCLIENTMODULE

Per prima cosa importare HttpClientModule in app-module e lo includiamo in imports.

Poi nel costruttore del service o nei moduli dove ci serve lo iniettiamo nel costruttore con il seguente comando:

private http: HttpClient

Per fare una chiamata dal client al server per prima cosa dobbiamo definire è l’url da chiamare. Ad esempio in questo modo:

private APIURL = ‘http://localhost:8000/users’

e poi la possiamo usare in un metodo nel seguente modo:

this.http.get(APIURL).subscribe(

data => console.log(data);

);

In questo caso, comunque abbiamo il problema del cross origin. C’è spiegato come risolverlo con lavarel. Se c’è questo problema vedere come risolverlo. Vedere come sono settate le autorizzazioni nel nostro progetto.

Una volta gestito il problema del cross origin ritorniamo alle chiamate asincrone:

this.http.get(this.APIURL).subscribe(

data => {

console.log(data);

},

error => alert(error.message)

);

return this.users;

Se scriviamo il codice in questo modo siccome abbiamo a che fare con una chiamata asincrona this.users è vuoto perhè prima che finisca la chiamata get users non avrà questi dati.

Per evitare quindi questo problema ritorniamo l’Observable direttamente nel seguente modo:

return this.http.get(this.APIURL);

e poi facciamo il subscibe nel client, nel component che richiama il metodo nel seguente modo:

this.service.getUsers().subscribe(

response => this.users =response.data;

);

Altre chiamate httpclient possono essere post (per modificare un utente)

return this.http.post(this.APIURL+’/’+user.id , user);

Per una cancellazione possiamo procedere nel seguente modo:

const data = {-method: ‘DELETE’};

return this.http.post(this.APIURL+’/’+user.id , data};

Creare schermate responsive vuol dire schermate che si adattano ai vari dispositivi. Con bootstrap si fa

VALIDAZIONE DI UN FORM

Per validare una form possiamo mettere nel button Save un attributo disabled che disabilita il button se il form nno è valido nel seguente modo:

<button [disabled]=”f.invalid” class= “btn btn-success” (click)=”saveUser()”>SAVE</button>

E poi possiamo mettere required per i campi del form richiesti

Poi per far uscire un messaggio quando il campo rimane vuoto nell’input del campo require possiamo aggiungere

#firstName=”ngModel” 🡪per indicare che firstName fa riferimento all’NgModel dell’imput che ci interessa e possiamo mettere un div con il messaggio che deve comparire utilizzando un ngIf come nel seguente esempio:

<div class=”alert-danger” \*ngIf=firstName.invalid>

Il campo first name è un campo richiesto

</div>

Da notare che in ngIf firstName viene inserito senza #

Un altro validatore invece di required è

Minlenght=”3” 🡪 che vuol dire che l’imput per essere valido deve avere un minimo di tre caratteri.

Se per un input mettiamo due validatori e vogliamo avere due messaggi diversi in base all’errore possiamo fare nel seguente modo.

Nel div con il primo messaggio \*ngIf=”firstName.errors.required”

E nel secondo div \*ngIf=”firstName.errors.minlenght”

E questi due div per non avere errori devono essere inseriti in un unico div che li contiene che avra un altro ngIf

<div \*ngIf=”firstName.invalid”>

Minlenght ha poi delle proprietà che si possono visualizzare con json e che possono essere richiamate.

firstName.dirty && firstName.touched && firstName.invalid 🡪 servono per indicare che il campo è invalido quando l’utente ha sporcato la casella di input e ci ha cliccato sopra

per la validazione si può usare poi maxlength o le reguar expression pattern = [0-9]{2,3} \_\_diciamo che il campo age può essere composto da numeri che vanno da 0 a 9 e può essere composto da 2 o massimo 3 cifre.

CREARE MODULO PER LE ROTTE

Quando si hanno molte rotte invece di inserirle in app.module si può creare un modulo apposito dove inserirle.

Ad esempio : app.routing-module

Che deve essere un unico file invece di una cartella

Const: routes: Routes= [

]

E và importato RouterModule.forRoot(routes) negli imports e poi dobbiamo esportare il Routing Module, quindi:

exports: [

RouterModule

],

in modo che l’applicazione possa fare riferimento a questo modulo.

paramMap serve per prendere una mappa di parametri

this.route.paramMap.subscribe(

(params) => {

This.user = this.userService.getUser(+params.id);

}

Quello che cambia è params che invece di utlizzare params.id utilizza ad esempio params.getAll() oppure params.get(‘id’) e per il resto poi non cambia.

COME PROTEGGERE LE ROTTE E DARE LE AUTORIZZAZIONI

Si può creare un component con un solo file come servizio e che si chiama route-guard-service.ts che serve per la protezione delle rotte e deve implementare CanActivate

In articolare se CanActivate ritorna true allora il component viene attivato altrimenti no. 🡪Lezione 51

Auth-service: servizio di autenticazione

<div class= offset-md-3 🡪 per centrare quello che c’è dentro

Email 🡪 validatore per email

Per predere un valore di un form si usa

form.value.NomeValore

!! 🡪 doppia negazione per ritornare una variabile booleana

Token 🡪 viene utilizzato per il login

JSON WEB TOKEN 🡪 uno standard per far comunicare in modo sicuro client e server. E’ il server che genera il token quando un client vuole autenticarsi e lo invia in modo criptato al client e poi quando il browser lo reinvia il server vede se lo ha in memoria se è o meno scaduto e gestisce meglio con gli header. Solo se si accede alla localStorage del dispositivo si può accedere ai dati.

PACKAGE jwt-auth 🡪per autenticazione lato server per web token

Guard 🡪 indica le protezioni

Guard => ‘api’ 🡪 per indicare che la guard di default è api

Aggiungere un AuthController per fare una chiamata di Login nel server e gestire l’autenticazione

Nei video usa LARAVEL per la parte di server. Vedere come farlo con STS

Vedere per lato cliente folder auth in progetto + metodo auth e login in apiservice

httpResponce è un tipo di risposta per cui possiamo avere dei valori tipo error per vedere il tipo di errore che ci ritorna la risposta.

httpErrorResponce 🡪 Serve per tornarci tutto il dettaglio dell’errore di una risposta di una chiamata

localStorage 🡪 la possiamo usare per prendere i dati dell’utente con il seguente metodo:

user = JSON.parse(localStorage.getItem("currentUser"));

Da notare se lo passiamo dal browser in JSON il metodo è

JSON.stringfy(……….)

Per prendere i dati JSON.parse

headers: {

"Authorization": ApiService.auth()

}

Per autorizzare le chiamate dal client verso il server e per passare il token tramite un headers

401 🡪 errore di autorizzazione col server

Nel component login mettiamo il comando router.navigate ([‘/’]) per far si che si venga ridirezionati nella pagina corretta dopo l’evento di login (emit).

canActivate: [RouteGuardService] viene messo in routing-module per indicare che una rotta non può essere visibile se non si ha l’autorizzazione

alert(….) serve per far comparire un messaggio a video

angular.json 🡪 sostituisce il file angular-cli.json vecchio e serve per impostare un progetto angular

in git -> clear serve per pulire lo schermo

Come aggiornare angular dalla versione 5.2 alla 6 🡪 lezione 69

Come installare typescript

Istruzioni per installare typescript sul sito: [www.typescriptlang.org/#download-links](http://www.typescriptlang.org/#download-links)

Comando da eseguire in cli è:

npm install -g typescript 🡪 -g indica installazione a livello globale

dopo l’installazione per vedere se è stato installato verifichiamo la versione con il seguente comando:

tsc -v

Typescript per funzionare necessita di Nodejs

Node è il nostro javascript lato server quindi per far partire un comando ts si può usare il comando in cli

Node nomeComponent.js

Vediamo una function con typescript:

function test() {

for (var i=0; i<10; i++){

console.log(i);

}

Console.log(i);

}

Test();

In questo caso come valore stamperà anche 10 perché la variabile la vede anche dopo il for. Se invece usiamo:

function test() {

for (let i=0; i<10; i++){

console.log(i);

}

Console.log(i);

}

Con let, dalla nuova versione di javascript del 2015, la variabile i avrà visibilità solo nel ciclo for ed il secondo consolo.log dà errore.

Se vogliamo compilare in typescript da riga di comando possiamo usare

tsc nomeComponent.ts

in questo caso viene creato il file js

se vogliamo cambiare il nome di un file ts da riga di comando possiamo farlo attraverso il seguente comando:

tsc index.ts –out index2.js

e, quindi troviamo index.js ed index2.js

Tipi di dati che possono essere gestiti con typescript:

A differenza di javascript che non prevede la tipizzazione in typescript si può usare la tipizzazione vediamo cosa cambia:

posso garantire che una variabile abbia solo il tipo di valore specificato senza incorrere in errore

es: varisFinish: boolean;

tipi primitivi: boolean, string, number, array, Object,

Sempre con typescript se scriviamo ad esempio:

name=’Nome’ senza segnalare il tipo ts in automatico riconosce che il valore di quella variabile è di tipo string e se poi faccio

name=5 🡪 avrò errore perché in automatico precedente alla variabile name è stato assegnato il tipo string

Gli array possono essere dichiarati come

Let months= [‘G’,’F’,’M’,…..];

è come se scriviamo

let months: string[];

un altro modo per rappresentare gli array è:

let months: Array<string> 🡪 questo modo è anche utile per rappresentare array di tipo generici.

Anche in questo caso posso cambiare l’array con valori dello stesso tipo

Let obj = {} 🡪 il tipo è un oggetto

Obj=5

Obj.toString() 🡪converte l’oggetto in stringa

Tupla -> E’ un array sempre fisso.

Vediamo un esempio:

Let params=[string, number];

params=[‘Hidran’,45]

function showData(pars:[string, number]){

console.log(pars[0]+’,’+pars[1]); 🡪 dove l’indice 0 indica string e indice 1 number

}

showData(params);

Con typescript quindi riusciamo e capire gli errori prima di eseguire il codice ma già quando lo scriviamo.

ENUM 🡪 Il tipo enum lo utilizziamo quando abbiamo dei codici che vogliamo utilizzare come codici parlanti ad esempio per i giorni della settimana.

Esempio:

enum DAYS{

LUN, MAR, MER, GIO,VEN,SAB

}

Console.log(DAYS.LUN); 🡪 ci ritorna come valore 0;

se scriviamo:

enum DAYS{

LUN=1,

MAR, MER, GIO,VEN,SAB,DOM

}

L’enum DAYS parte da 1 invece che da zero e gli altri via di seguito.

Console.log(DAYS.LUN); 🡪 ci ritorna come valore 1;

Console.log(DAYS[3]); 🡪 ci ritorna come valore MER;

Quindi agli enum possiamo accedere o in un modo o in un altro.

Esempio di come confrontare due enum:

function isWorkingDay(day: DAYS){

if(day === DAYS.SAB || day === DAYS.SAB){

return true;

}

Return false;

}

Console.log (isWorking(DAYS.MER)); 🡪 ritorna false;

se mettiamo:

Console.log (isWorking(3)); 🡪 ritorna false perché è un giorno della settimana(MER);

ma anche se mettiamo:

Console.log (isWorking(8)); 🡪torna false ma poi non viene assegnato nessun enum.

Bisogna evitare la sovrascrittura nelle enumerazioni. Ad esempio se sappiamo che enum DAYS che abbiamo creato in un file arriva ad 8 noi possiamo decidere di aggiungere ultriori valori all’ENUM anche in un altro file ma accertandoci di avere la seguente forma:

enum DAYS{

LAA =8,

NAA,

}

In questo modo evitiamo che LUN e LAA abbiano un unico indice di accesso=1

Vediamo, ora gli altri tipi di dati typescript che sono:

any, void, null, undefined e never

any 🡪 vuol dire che il valore può essere di qualunque tipo. Ad esempio se mettiamo 45 a numero poi possiamo mettere una stringa

any si usa quando i dati ci arrivano da una chiamata AJAX

con un array per indicare che posso assegnare ad una variabile un array che può avere qualunque tipo di dato al suo interno. E questo si utilizza molto in typescript.

Void 🡪 non torna nulla e può essere utilizzato nella function quando non ritorna nulla

function printName() : void{

console.log(‘erro’)

}

In realtà poi questa funzione torna null o undefined perché a void si può assegnare solo undefined o null.

Let myAge: number; 🡪 in questo caso qui la variabile è di tipo undefined.

A differenza di void che torna vuoto il tipo never non torna mai nulla perché solleva un errore

Interface in typescript:

vengono utilizzate per creare dei tipi di dati. Esempi di interface sono quelli utilizzati nei nostri model.

In una interfaccia possiamo avere dei dati obbligatori e dei dati opzionali. Per dire che alcuni dei dati sono opzionali si mette il punto interrogativo subito dopo il nome della variabile

Esempio:

interface Iuser{

lastName : string,

firsName : string,

addres? : string,

[propName : string] : any 🡪 questo si può utilizzare quando l’interfaccia viene utilizzata nelle chiamate AJAX per indicare che se ad esempio vengono aggiunte nuove colonne al DB in automatico possono essere prese nel front end senza problemi ed errori. Diciamo che questa interface può ricevere altri valori. Se però siamo sicuri dei dati che vogliamo che ci vengano passati allora questa proprietà non la mettiamo e definiamo per bene tutte le variabili dell’interface

}

In questo caso i primi due sono campi richiesti per quello specifico tipo di dato Iuser mentre l’ultimo può essere o meno presente.

Funzioni. Quelle generali le abbiamo già viste in precedenza.

ArrowFunction. Esempio:

let myArrowFunc = (e: any) => { console.log(e) }

myArrowFunc(‘Ciao’);

let myArrowFunc = (e: number) : number => { e\*2} 🡪 sto dicendo che alla funzione passo un number e e che voglio che mi ritorni un number quando richiamo la funzione

Vediamo la visibilità dei metodi di una classe Typescript:

tutte le proprietà di un oggetto class se non specifico di default sono bublic. Ciò vuol dire che ci posso accedere e modificare tranquillamente la proprietà non solo dall’interno della classe ma anche dall’esterno.

Se una proprietà la metto come protected allora per accedere a quella proprietà devo avere i metodi set e get anche da una classe che estende quella classe ma non ci posso accedere direttamente.

Se una proprietà è d tipo private neanche la classe che estende ci può accedere con i metodi get e set ma solo la classe in cui viene dichiarate può accedere e con metodi get e set.

Quindi in base a quello che vogliamo fare con le nostre proprietà impostiamo la visibilità della proprietà.

Come estendere una classe base.

Anche con typescript è possibile ereditare da una classe base con extends

In una classe che eredita dalla classe padre possiamo inserire anche il costruttore. Se lo inseriamo, però, nel costruttore dobbiamo inserire “super” per indicare che deve essere chiamato per forza il costruttore della classe parent. Per fare questo nel costruttore figlio mettiamo:

constructor: (nomeParametroParent: tipo){

super(nomeParamentroParent);

…….

……

}

Super nel costruttore va messo prima di impostare qualunque altra proprietà se no dà errore.

Quando si estende una classe si può fare anche l’override di un metodo parent perché ad esempio ci serve quel metodo modificato in modo che ci possa restituire altro.

Proprietà readOnly 🡪

readonly version: string =’version 1.1’

oppure solo con il costruttore. Poi non posso più modificare questa proprietà perché la proprietà è ReadOnly. Con il costruttore, poi possiamo impostare, poi, la visibilità delle altre proprietà dichiarandole private o protected direttamente quando le passiamo dentro il costruttore. 🡪 Questo modo per impostare la visibilità è molto usato.

I metodi get e set si utilizzano per le variabili che vengono dichiarate private in una classe.

Vediamo un esempio:

public class User{

private \_name:string;

set name(name: string){

this.name=name.toUpperCase();

}

Get \_name(){

Return name;

}

}

Per richiamare, poi, la proprieta possiamo scrivere

Let user =new User();

user.name = ‘Hidran’;

console.log(user.name);

in questo modo vengono richiamati i metodi getter e setter della classe User.

Vediamo ora un interfaccia che estende una classe.

Class MyLogger {

Log ( msg : string):void {

Console.log(msg);

}

generateId (): number {

return Math.round(Math.random\*100000)

}

}

Interface MyLog extends MyLogger{

Email : string;

} 🡪 In questo caso l’interfaccia avrà solo la dichiarazione dei metodi: Log ( msg : string):void e

generateId (): number ed una variabile email.

Quindi in typescript esiste questa possibilità

Classi statiche:

Le proprietà statiche che creiamo all’interno di una classe fanno parte della classe stessa e non dell’istanza della classe. Vediamo un esempio:

class MathCalc{

static readonly PI=3.14;

static calcCirclePerimeter(r: number){

return 2\*MathCalc\*r;

}

Let calc = new MathCalc();

console.log(calc.PI) 🡪 restituisce undefined. Per poterla usare dobbiamo scrivere

console.log(MathCalc.PI) 🡪 ed in questo caso restituisce il valore PI

console.log(calcCirclePerimeter(2)); 🡪 anche in questo caso devo utilizzare la classe

Anche se siamo all’interno della classe e di un metodo che non è statico se usiamo variabile statica la dobbiamo mettere con il nome della classe.

Classe astratta : è una classe in cui viene implementato qualche metodo ma gli altri vengono solo dichiarati aspettando che, poi, vengano implementati dalla classe che la implementa (metodi astratti 🡪 deve averne almeno uno.

Abstract class Logger{

Abstract log (msg : string) : void

generateId() : number {

return Math.round(Math.random\*100000)

}

}

Class ConsoleLogger extends Logger{

log (msg : string) : void{

console.log(msg);

}

}

Let log = new ConsoleLogger() 🡪 così funziona invece Let log = new Logger() dà errore.

Una interfaccia ci serve solo per dire ad una classe i metodi che questa deve avere senza alcuna implementazione. Invece la classe abstract ha dei metodi implementati e lascia alla classe che la estende il compito di implementare i metodi non implementati. Una classe può implementare un interfaccia ed estendere una classe.

MODULES

Un modulo è un file di javascript il cui contenuto è isolato cioè non và a sporcare l’ambiente globale di javascript e ci permette di dividere il progetto in moduli ed utilizzare solo alcune parti del modulo nascondendone altre. Per utilizzarlo utilizziamo l’import nei moduli dove lo vogliamo importare.

Nell’import ‘./book/nomr’ 🡪 il punto prima dello slash serve ad indicare che il modulo si trova a partire dal folder book

Export va messo nelle variabili o alla classe o ai metodi che si vogliono esportare.

* Per dettagli lezione 87

Per poter utilizzare i moduli si devono usare dei loader module. Ad esempio nodejs ha già il suo loader ed utilizza required.

PROMISE

Le promise ci permettono di gestire i processi asincroni in un modo più sinrono e rappresentano un valore che possiamo gestire nel futuro nel senso che se ho una chiamata asincrona la promise ci rappresenta il valore che questa chiamata avrà in futuro ma può darsi anche che non lo ritorni. Il valore di ritorno di una promise è immutabile anche se abbiamo diversi listeners sulla chimata. Il valore poi è sempre disponibile anche se registriamo l’handler dopo che la promise è stata risolta. Per utilizzarla si procede nel seguente modo:

var p =new Promise(function(resolve,reject){

if (condizione)

resolve(value) 🡪 va a buon fine

else

reject(reason) 🡪 errore, rifiutata

}

Quindi nella new viene passata una funzione chiamata risolve che riceve due parametri: un parametro che serve per risolvere la promise e l’altro per rifiutarla.

Da lezione 89

La promise si può trovare in tre stati:

* Pending(Pendente): pendente quando la promise non è stata ancora risolta o rigettata perché non ha ritornato ancora nessun valore;
* Fullfilled(risolta): quando viene chiamato il primo metodo resolve
* Rejected: quando viene chiamato il secondo metodo perché è stata rifiutata.

90 🡪 da vedere per la gestione degli oggetti json e 91 chiamate asincrone