**ANGULAR**

La cartella del progetto è all’interno di /src, i componenti del progetto sono all’interno di /src/app.

Il file app.module.ts dentro /src/app contiene tutti i moduli utilizzati nel progetto (è da questo file che si inseriscono nuovi moduli/componenti).

**Component**

Componente base di un’applicativo.

**----- Creazione nuovo componente -----**

Per creare un nuovo componente bisogna aggiungere due file, uno .ts ed uno .html. (Come regola base per la nomenclatura dei file di solito si usa la forma “nomeComponente”.component.ts/.html).

Il file .html non richiede nulla in particolare per essere un componente mentre il file .ts richiede una certa struttura.

Esempio del componente server

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

@Component({

selector: 'app-server',

templateUrl: './server.component.html'

})

export class ServerComponent {

}

Per prima cosa un Component è semplicemente una classe TypeScript con un tag @Component per identificare che è un componente.

All’interno del tag @Component vanno specificati:

**Selector (non obbligatorio) ->** il tag che va utilizzato negli altri componenti per poter includere questo componente (nell’esempio **app-server**). Il tag può essere inserito in varie forme:

**selector: 'app-servers'** -> Per aggiungere il nostro component in un file html lo dobbiamo inserire come tag. e.g. <app-servers></app-servers>. (**Il più usato**)

**selector: '[app-servers]'** -> Per aggiungere il nostro component in un file html lo dobbiamo inserire come attributo. e.g. <div app-servers></div> (non deve essere obbligatoriamente div)

**selector: '.app-servers'** -> Per aggiungere il nostro component in un file html lo dobbiamo inserire come classe. e.g. <div class="app-servers"></div>

**templateUrl/template (obbligatorio) ->** Il path relativo per il template associato al nostro componente nel caso di **templateUrl**, oppure direttamente il codice html nel caso di **template** (il codice va inserito tra apici ‘ nel caso di una sola riga oppure tra backquote nel caso di più righe)

**styleUrls/styles (non obbligatorio) ->** Gli stili css da applicare al nostro componente. Nel caso di **styleUrls** vanno inseriti i path dei file css, nel caso di **styles** invece va inserito direttamente il codice css come sopra per **template**.

**Providers (non obbligatorio)** -> array che contiene tutti i servizi utilizzati dal componente

Una volta generati i due file del componente bisogna aggiungerlo dentro il file **app.module.ts** per indicarne ad angular la sua presenza.

Per includerlo basta aggiungere dentro l’array **declarations** il nome della nostra classe (nell’esempio **ServerComponent**) ed importarla (e.g. **import { ServerComponent } from './server/server.component';** ).

E’ possibile inoltre generare automaticamente un component tramite il relativo comando per CLI (vedi sotto in **Comandi CLI**).

**Databinding**

**One way databinding**

**String** **interpolation** (OUTPUT) -> {{ stringa }} -> Pone all’interno delle parentesi una stringa, questa stringa può essere fornita tramite una variabile (sia da una stringa che da un qualsiasi valore facilmente convertibile in stringa), da una funzione che ritorna una variabile (con gli stessi attributi di prima) o direttamente da una stringa hardcoded.

**Property** **binding** (OUTPUT) -> [disabled] = “value” -> Assegna alla proprietà il valore value. Tale valore può essere inserito tramite variabile oppure tramite risultato di una funzione. (nell’esempio la proprietà disabled collegata ad un button consente di attivarlo o disattivarlo in base al valore di value.

E.G. posta var = “PROVA” -> <p [innerText] = “var”></p> == <p>{{ var }}</p>

Nel primo caso viene usato il **property** **binding** bindando la proprieta innerText (sarebbe il testo interno) di **p** con la variabile **var**, nel secondo caso viene usata la **string** **interpolation** inserendo come testo il contenuto di **var.**

**Event binding** (INPUT) -> (evento) = “funzione($event)” ($event è opzionale) -> Binda l’avvenimento di un determinato evento con una funzione. Quando l’evento viene triggerato automaticamente viene eseguita la funzione. Il parametro $event contiene tutte le informazioni relative all’evento, per esempio, nel caso di un input, contiene il valore inserito.

(e.g. 1 (input)="onUpdateServerName($event)" -> quando si inserisce testo viene chiamata la funzione ed aggiornato il server name con quello che l’utente inserisce)

(e.g. 2 (click) = “onCreateServer()” -> quando l’oggetto viene cliccato parte la funzione onCreateServer.

**Two way databinding**

[( ngModel )] = “variabile” -> Binda la variabile con il contenuto dell’elemento su cui è posto. Se la variabile ha qualche valore l’elemento prende il valore, se l’elemento viene modificato la variabile viene modificata a sua volta.

(e.g. [(ngModel)] = "serverName" -> Questo codice è inserito in un input, all’avvio il testo dell’input diventa quello contenuto nella variabile serverName, quando il testo dell’input viene modificato anche il contenuti di serverName viene modificato a sua volta.

**Local Reference**

E’ possibile creare una **local reference (#nome)** negli elementi all’interno del file html. Praticamente è una stringa identificativa dell’elemento che possiamo utilizzare all’interno della struttura html (solamente nella struttura html). e.g. **<input type=”text” #TEST>** in questo caso all’interno del file html dove abbiamo posto questa input possiamo riferirci ad essa utilizzando la stringa **TEST**, per esempio nel caso dovessimo utilizzare l’**event binding** possiamo passare come parametro direttamente **TEST**. e.g. **<button (click)=”function(TEST)”>CLICK</button>**, in questo caso alla funzione **function** verrà inviato un parametro di tipo **HTMLInputElement**.

**Accesso al DOM** -> **@ViewChild( selettore) var: ElementRef;** -> ElementRef va importato da @angular/core. Tramite **ViewChild** è possibile ottenere elementi del DOM specificati attraverso il selettore. Solitamente come selettore si passa la **Local Reference** dell’elemento che vogliamo ottenere.

Per accedere all’elemento dobbiamo utilizzare **var.nativeElement** che contiene le informazioni dell’elemento (nel caso stiamo accedendo ad un **input** il **nativeElement** sarà di tipo **HTMLInputElement**).

(N.B. E’ consigliato non andare a modificare i valori di un elemento tramite l’accesso diretto al DOM, esistono delle vie migliori tipo **string interpolation**).

**Directives**

**Attribute directives:**

**ngStyle** -> Serve per modificare lo stile css dell’elemento. (e.g. **[ngStyle]="{ backgroundColor: getColor() }”**, vanno inserite le parentesi quadre poichè questa directive viene utilizzata solo con il property binding, dentro le graffe abbiamo la proprietà **backgroundColor** ed il valore **getColor()** che in questo caso è il risultato di una funzione).

**ngClass** -> Serve per assegnare dinamicamente una classe ad un elemento. (e.g. **[ngClass]="{ online: serverStatus === 'online' }"**, vengono messe le quadre per lo stesso motivo di **ngStyle**. In questo esempio viene assegnata all’elemento la classe **online** se la variabile **serverSatatus** ha valore **online**.

**Creazione custom directive** -> E’ possibile creare una **attribute directive personale** tramite il tag **@Directive**, praticamente il funzionamento è uguale a quello della creazione di un **component** solo che al posto del tag **@Component** si ha **@Directive**. Quest’ultimo prende come parametro il **selector** con il quale verrà poi richiamata la **directive** (il selettore è utile metterlo tra [] così da poterlo assegnare come proprietà ad un tag. e.g. preso un selettore **‘[highlight]’** posso assegnarlo ad un paragrafo tramite **<p highlight></p>** mettendolo come proprietà.

Per poter ottenere l’elemento a cui è stata assegnata la nostra **directive** possiamo richiedere al costruttore delle informazioni -> **constructor(private elRef: ElementRef, private renderer: Renderer2) { }** -> in questo caso Angular ci fornisce l’elemento a cui siamo attaccati (**elRef**) e possiamo richiedere un oggetto di tipo **Renderer2** per andare ad effettuare le modifiche (volendo potremmo farle direttamente tramite elRef ma non è consigliato).

**@HostListener( evento ) function() { }** -> E’ utilizzato per bindarci ad un evento dell’elemento. Messo dentro una nostra **directive** si binda all’elemento su cui l’abbiamo inserita. Il campo **evento** indica un avvenimento (e.g. mouseenter, mouseleave, click,...), la **function** invece è quello che viene lanciato. Volendo possiamo anche mettere un parametro di tipo **event** dentro la **function** per ottenere informazioni sull’evento.

**@HostBinding( proprietà ) nome : String = valore;** -> Simile ad **HostListener** ma per le proprietà dell’elemento (è un ulteriore metodo per modificare lo stile dell’elemento). Al posto di **proprietà** possiamo inserire cosa andare a modificare dell’elemento (e.g. **style.backgroundColor** -> importante camel case). In questo modo ci basta assegnare il valore alla variabile per andare a modificare quella proprietà nell’elemento (e.g. creando una var **test** bindata a **style.backgroundColor** andando ad assegnargli **“blue”** cambierà il colore di background in blu. N.B. La variabile deve essere inizializzata con un valore poichè altrimenti darebbe errore)

**Structural directives:**

L’asterisco nelle directives indica che è una structural directive che modifica il template del DOM.

Non è possibile inserire due o più **structural directives** nello stesso elemento

**\*ngIf = “serverCreated”** -> Se serverCreated è true l’elemento su cui è posta viene creato ed inserito nel DOM, in caso contrario l’elemento non è presente. Al contrario di una property **disabled** in questo caso l’elemento viene generato o tolto dal DOM mentre con disabled l’elemento resta presente nel DOM ma viene nascosto e mostrato.

**\*ngFor = “let var of vars”** -> Viene posto dentro un elemento e lo replica per ogni variabile contenuta dentro vars. (e.g. **<app-server \*ngFor="let server of servers"; let i = index></app-server>**, app-server viene inserito per ogni server presente dentro servers, la parte **let i = index** è opzionale e consente di avere sulla variabile **i** l’indice del **for**)

**[ngSwitch] = “value” / \*ngSwitchCase = “val” / \*ngSwitchDefault** -> Sono tre structural directive che consentono di sostituire una catena di **ngIf**. Per prima cosa va creato un contenitore con dentro **[ngSwitch]=”value”** dove **value** è il valore su cui dobbiamo switchare. Dentro il contenitore possiamo inserire una serie di elementi che contengono **\*ngSwitchCase=”val”** dove **val** è un preciso valore (che potrebbe essere preso da **value**. Nel caso **value** abbia uno di quei valori viene mostrato l’elemento relativo, nel caso non ci sia nessun **val** corrispondente a **value** non viene mostrato nulla. Se fosse presente un elemento con **\*ngSwitchDefault** verrebbe invece mostrato quello.

**ngContent** -> Posto all’interno del file html di un component consente di inserire del contenuto all’interno dei tag html quando viene utilizzato. e.g preso un component **test** con selector **app-test** ed un file html composto solamente da **<ng-content></ng-content>**, possiamo all’intero di un nostro file html inserire un paragrafo all’interno dei tag **app-test** come segue **<app-test><p>Ciao</p></app-test>**. In questo modo il contenuto al loro interno non verrà eliminato ma mostrato, nel caso fosse omesso **ng-content** dentro il componente **test** il contenuto sarebbe stato eliminato di default.

**Databinding Custom**

**Binding a Custom Property ( @Input( alias ) var )** -> Ponendo il costrutto @Input() (da includere nel file Typescript) davanti ad una variabile contenuta all’interno del nostro component possiamo indicare che tale proprietà è accedibile dall’esterno. In seguito tramite il **property binding** visto precedentemente è possibile andare ad assegnare valori a quella specifica proprietà. Il valore **alias** (opzionale, in caso di omissione diventa **var**) indica il nome della proprietà che dovremo inserire nel **property binding** mentre il valor **var** indica il nome con il quale potremo chiamare la variabile nel file TypeScript.

(e.g. Abbiamo un component **test** con selettore **app-test** e variabile **@Input(alias) var: string;** . Dentro un nostro html possiamo inserire **<app-test [alias]=”prova” ></app-test>**. In questo modo verrà incluso il component **test** dentro il nostro html e verrà assegnata alla variabile **var** il valore **prova**).

**Binding a Custom Event** ( **@Output( alias ) var = new EventEmitter<TYPE>(); )** -> In questo modo è possibile creare un custom event all’interno del nostro component. All’interno del codice dove è definito **var** possiamo utilizzare la funzione **var.emit(DATA)** per inviare avviare l’evento ed inviare i **DATI** inseriti.

Il tipo **EventEmitter** richiede di specificare il tipo (**TYPE**) dei dati da inviare. All’interno del file html dove inseriamo il nostro component possiamo, tramite l’**event binding** visto precedentemente, segnarci all’avvenimento del dato evento. Sempre come visto precedentemente possiamo assegnargli una funzione da far partire al suo avvenimento e passargli come parametri **$event** per ricevere i **DATI** inviati dal component. Il valore **alias** è opzionale come sopra ed in caso di omissione agisce allo stesso modo.

(e.g. Abbiamo un componente **test** con selettore **app-test** ed evento **@Output( alias ) var = new EventEmitter<{flag: string}>();** avviato tramite **this.var.emit({ flag: “prova” });**. Dentro un nostro html possiamo inserire **<test (alias)=”function($event)”></test>**. In questo modo quando l’evento verrà attivato (**emit**) dal componente **test** il component dove abbiamo inserito **app-test** attiverà la funzione **function** ricevendo all’interno della variabile **$event** i dati inviati da **emit**).

**Services**

Per creare un servizio non serve costrutti particolari di angular ma basta typescript (in alcuni casi non basta).

Un servizio è praticamente una classe typescript che mette a disposizione dei metodi per gli altri elemento del progetto.

Per includere un servizio all’interno di un component/directive bisogna aggiungere delle cose:

1. Per prima cosa bisogna importare il servizio che si vuole includere
2. Successivamente va inserito dentro il campo **providers** del descrittore (e.g. **component/directive**) il nostro servizio (e.g. **providers: [LoggingService]**). Il campo è formato da un array visto che si possono inserire vari servizi in uno stesso componente.
3. Infine bisogna avvertire Angular di effettuare l’inject del servizio all’interno del nostro componente. Per farlo basta inserire un parametro nel costruttore con lo stesso tipo del nostro servizio (e.g. **constructor(private logServ: LoggingService)** in questo esempio definiamo la variabile **logServ** di tipo **LoggingService**. Sarà poi Angular ad occuparsi di effettuare l’inject del servizio all’interno della variabile e rendercela disponibile)

Una volta effettuati i tre punti è possibile utilizzare tutte le funzionalità del servizio andando ad utilizzare la variabile che abbiamo definito.

I servizi seguono uno schema gerarchico. Se ho un gruppo di tre componenti uno dentro l’altro basta che inserisco il servizio nei providers del primo per inserirlo in tutti gli altri (ovviamente devo comunque effettuare i punti 1 e 3).

Nel caso in cui però andassi ad inserire il servizio su uno dei provider interni si genererebbe una nuova instanza del servizio stesso. In pratica il servizio che utilizzo nel padre ed il servizio che utilizzo nel figlio sarebbero due instanze diverse, rendendo impossibile una comunicazione tra diversi livelli di componenti.

Se inserisco il servizio nei provider di **AppModule** automaticamente tutti i component lo ricevono.

E’ possibile inserire un servizio dentro un altro servizio, in questo caso bisogna inserire il tag **@Injectable()** sopra la definizione della classe (Come ad esempio per i tag **@Component()** o **@Directive()**). Una volta inserito basta solo definire al solito il costruttore ed importarlo. L’utilizzo è uguale agli altri casi. (N.B. Per i servizi in questione devono essere **obbligatoriamente** inseriti dentro il **providers** di app.module altrimenti non funziona)

**Routing**

Il routing serve per consentire la navigazione tra diversi url all’interno del nostro applicativo (e.g. /home, /users...).

Le informazioni base del routing vanno inserite dentro app.module poichè è la root della nostra applicazione.

All’interno di app.module andremo ad inserire i **Routes**. e.g. :

const appRoutes: Routes = [

{ path: '', component: HomeComponent },

{ path: 'users', component: UsersComponent },

{ path: 'users/:id/:name', component: UsersComponent },

{ path: 'servers', component: ServersComponent },

];

In questo caso abbiamo definito le path **/ /users /servers** collegandole rispettivamente a **HomeComponent**, **UsersComponent** e **ServersComponent**.

La path **/users/:id/:name** invece definisce una path con parti dinamiche, il **:** indica ad angular che il contenuto sarà dinamico e dovrà essere “salvato” con identificativo **id**. e.g. possiamo avere **/users/20/Paolo** dove con **id** avremo **20** mentre con **name** avremo **Paolo**.

Per ottenere questi valori su typescript dovremo effettuare l’inject di una variabile di tipo **ActivatedRoute** e tramite la sua funzione **.snapshot.params** potremo andare a recuperare i valori così da poterli usare. (e.g. **this.route.snapshot.params['id']** -> in questo caso prendiamo il valore di **id**).

E’ possibile anche creare un routing annidato dentro un altro routing utilizzandoil tag **children**.

L’esempio di prima esteso

const appRoutes: Routes = [

{ path: '', component: HomeComponent },

{

path: 'users', component: UsersComponent, children: [

{ path: ':id/:name', component: UserComponent }

]

},

{

path: 'servers', component: ServersComponent, children: [

{ path: ':id', component: ServerComponent },

{ path: ':id/edit', component: EditServerComponent }

]

},

];

In questo caso abbiamo due routing annidati, uno dentro **users** e l’altro dentro **servers**. Ovviamente dovremo mettere dentro i file html dei rispettivi component l’elemento **<router-outlet></router-outlet>** per indicare dove vogliamo caricare i component figli.

Successivamente va importato dentro **imports** di **app.module** il modulo **RouterModule** richiamando la funzione **.forRoot()** e passandogli come parametro le nostre **Routes** (e.g. **RouterModule.forRoot(appRoutes)** ).

Una volta definite le nostre **routes** dobbiamo decidere dove mostrare le informazioni. All’interno del file html dove vogliamo mostrarle dobbiamo inserire il tag **<router-outlet></router-outlet>** che consentirà la visualizzazione delle varie pagine in base a ciò che abbiamo definito sopra.

Per la navigazione tra le **routes** non si usa l’attributo **href** (poichè eseguirebbe un aggiornamento della pagina) ma si usa l’attributo Angular **routerLink**. Praticamente basta mettere **routerLink=”/path”** (e.g. **routerLink=”/servers”**) ed angular gestirà il routing dell’elemento senza aggiornare la pagina. Come path è possibile mettere percorsi assoluti o relativi (e.g. **/path, path, ../path**)

Per effettuare la navigazione tra pagine utilizzando typescript basta effettuare l’inject del **Router** nel costruttore (ovviamente bisogna importarlo). Poi richiamare la funzione **navigate** ed inserire i vari **path**. e.g. **this.router.navigate(['/servers']);** -> in questo caso si effettua la navigazione verso l’indirizzo assoluto **servers**, è possibile inserire altri path **[‘/servers’,’test’]** in questo caso si andrebbe su **/servers/test**. Per effettuare la navigazione verso un indirizzo relativo bisogna però specificare anche in relazione a cosa aggiungendo il path della posizione attuale e.g. **this.router.navigate(['servers'], { relativeTo: this.route });** - > In questo esempio viene inserito il path relativo **this.route** che ci indica in base a cosa dobbiamo navigare. Per ottenere il path basta effettiare l’injection nel costructor di una variabile di tipo **ActivatedRoute** che bisogna importare da **@angular/router** come **Router**.

E’ possibile passare anche dei parametri di query oppure dei fragment.

Nel caso dei parametri di query bisogna:

* Nel caso lo stiamo effettuando su un’attributo (e.g. routerLink) bisogna aggiungere la proprietà **[queryParams]** ed assegnargli un array json con il suo valore. e.g. **[queryParams]="{allowEdit: '1'}"**
* Nel caso lo stessimo aggiungendo via typescript basta aggiungeretra i parametri **queryParams** ed assegnargli un array json come prima e.g. **{ queryParams: { allowEdit: '1' }}**.

Per il **fragment** bisogna fare la stessa cosa di **queryParams** solo che non prende un array json poichè al fragment può essere assegnato un solo valore, quindi prende una stringa in tutti e due i casi

e.g. **<a [routerLink]="['/servers', 5, 'edit']" [queryParams]="{allowEdit: '1'}" fragment="loading">** / **this.router.navigate(['/servers', id, 'edit'], { queryParams: { allowEdit: '1' }, fragment: 'loading' });**

**Observable/Observer (rxjs)**

**Observable** -> Elemento a cui possiamo sottoscriverci per ricevere update

**Observer** -> Elemento che invia informazioni a chi è sottoscritto

I risultato di un **Observable** sono di 3 tipi diversi

* Dati -> Praticamente i dati grezzi che vengono inviati dall’**observer**
* Errore -> Caso terminale di errore (viene inviato un messaggio)
* Completato -> Caso terminale di completamento con successo dell’**observer** (non viene inviato nulla, solo notificato)

Un **observer** viene costruito nel seguente modo:

myObservable = Observable.create(

(observer: Observer<string>) => {

setTimeout(() => {

observer.next('first package');

}, 2000);

setTimeout(() => {

observer.complete();

}, 5000);

setTimeout(() => {

observer.error('third package');

}, 6000);

}

);

Nell’esempio possiamo vedere una definizione di un **observable** (in questo caso observer).

Nella definizione dell’**Observer** (**Observer<type>**) va specificato il tipo dei dati che verrano restituiti (nel caso i dati inviati con **.next** e **.error** siano di un tipo diverso angular non compilerà)

La funzione **.next( data )** ci consente di inviare i dati grezzi (dati opzionali)

La funzione **.complete()** di notificare il completamento

La funzione **.error( data )** di notificare l’errore (dati opzionali)

Per sottoscriversi ad un **observer** basta utilizzare la funzione **subscribe**

myObservable.subscribe(

(data: string) => {

console.log(data);

},

(error: string) => {

console.log(error);

},

() => { console.log('completed'); }

);

In questo caso viene utilizzata la funzione **subscribe** sull’observable di tipo **myObservable**. Dentro al subscribe vanno definite tre funzioni (solo la prima obbligatoria, le altre sono opzionali). La prima gestisce la ricezione dei dati grezzi, la seconda la ricezione dell’errore mentre la terza il completamento dell’observable.

Un’altro oggetto molto utile è il **Subject** che praticamente è un **observer** ed **observable** allo stesso tempo.

Una volta definita una variabile di tipo subject (e.g. **userActivated = new Subject();** ) è possibile usare tramite essa sia le funzioni dell’**observer** che sottoscriversi ad essa.

Quindi possiamo utilizzare **.next .error .complete** per inviare le informazioni ed usare **.subscribe** per sottoscriverci (gli EventEmitter di angular sfruttano lo stesso oggetto).

**Forms**

**Template forms**

Per creare una form bisogna utilizzare il tag di default di html **form**.

Il tag però non presenterà le funzioni standard di javascript ma dovrà essere esteso con le funzioni di angular.

Ci sono due modi per creare ed utilizzare una form:

* **<form (ngSubmit)="onSubmit()" #f="ngForm">**
* **<form (ngSubmit)="onSubmit(f)" #f="ngForm">**

La parte importante dei due medoti è **#f=”ngForm”** in cui viene generata la local reference e specificato che è una **form angular**. Nel primo caso viene inviata poi alla funzione **onSubmit** nel secondo caso invece viene presa tramite il costrutto **ViewChild** visto in precedenza (**@ViewChild('f') signupForm: NgForm;** ). Il vantaggio del secondo metodo rispetto al primo sta nel momento in cui possiamo utilizzarlo, nel primo caso avremo accesso alla form solo una volta che verrà richiamata la funzione **onSubmit()**. Nel secondo metodo invece, avendo creato la variabile, avremo subito accesso alla form già dall’inizializzazione della stessa.

Una volta generata la form bisogna preparare i campi al suo interno per la gestione tramite Angular.

Per abilitare un campo alla gestione tramite angular bisogna inserire al suo interno **ngModel** e.g. **<input type="text" id="username" class="form-control" ngModel name="username" required>**. Inoltre bisogna anche definire il nome del campo che verrà poi utilizzato per associare tutti i vari valori ottenuti.

All’interno della definizione dei campi possiamo aggiungere anche altri costrutti. Ad esempio:

* **Required** -> utilizzato per indicare che quel campo è necessario e che in caso di mancanza non è valida la form
* **Email** ->Utilizzato per indicare che il campo contiene una mail. Grazie a questo tag angular si occupa di controllare che la mail sia valida, in caso contrario avverte che il campo è invalido.
* **...** ne esistono molti altri, da guardare sul sito di angular (sono sotto il nome di directives per le form).

Per il submit della form bisogna inserire un button con type submit, in automatico angular farà partire la funzione associata (in questo caso **onSubmit()**).

All’interno della variabile di tipo **NgForm** indicata precedentemente (ViewChild o quella passata alla funzione) ci sono varie informazioni come: i valori della form, il fatto se la form è valida o invalida (se un solo campo non è valido la form non è considerata valida), le informazioni su ogni singolo campo (tipo validità,...), se la form è stata modificata, se la form è stata cliccata e varie altre informazioni.

Angular inoltre assegna anche delle classi ai campi della form modificandole dinamicamente in base agli eventi. Queste classi indicano se il campo è stato modificato, toccato oppure anche se è valido o invalido (molto utili per identificare se qualche campo non è corretto e notificarlo in tempo reale -> per esempio mettendo uno style css sui campi invalidi e che sono stati modificati).

E’ possibile ottenere questi valori anche al livello di html ma per fare ciò bisogna indicare una **localReference** ed associargli **ngModel**. e.g. **<input type="email" id="email" class="form-control" ngModel name="email" required email #email="ngModel">,** ad esempio in questo caso di campo email è stata creata la local reference email ed assegnato **ngModel** per aggiungere le informazioni di **form** come **email.valid** o **email.touched** utili per creare un paragrafo da visualizzare se il contenuto dell’input non è valido e.g. **<span class="help-block" \*ngIf="!email.valid && email.touched">Please enter a valid email!</span>**. E’ possibile in questo modo anche disattivare il button di submit, tramite la local reference della form, **f** nel caso dell’esempio di prima, possiamo accedere alla proprietà **valid** tramite **f.valid** che ci restituisce **true** nel caso che la form sia valid, **false** altrimenti (è possibile accedere alla proprietà solo perchè abbiamo associato alla local reference **f** **ngForm**).

Bindando **ngModel** solo una volta è possibile inserire un campo di default. Ad esempio nel campo username visto sopra potremmo inserire un username di default tramite **[ngModel]=” ’Username’ ”.**

Angular offre anche la possibilità di generare dei sottogruppi di campi all’interno di una form. Per fare ciò basta inserire delle proprietà angular all’interno di una **div** che contiene i nostri campi.

**<div id="user-data" ngModelGroup="userData" #userData="ngModelGroup">** -> **ngModelGroup** identifica che la div contiene un gruppo di informazioni e che questo gruppo verrà identificato con **userData** inoltre come per la **ngForm** bisogna creare la local reference del gruppo (in questo caso **userData**) ed assegnargli però **ngModelGroup**.

E’ possibile anche inserire dei valori dentro la form tramite typescript utilizzando la variabile di tipo **NgForm** ottenuta precedentemente tramite **ViewChild**.

Ci sono due modi per modificare i valori:

* Il primo consente di modificare obbligatoriamente tutti i valori (non è possibile modificare un solo valore ma siamo costretti a modificarli tutti).

this.signupForm.setValue({

userData: {

username: suggestedName,

email: ''

},

secret: 'pet',

questionAnswer: '',

gender: 'male'

});

Con questo metodo bisogna passare alla funzione **setValue** il json di tutta la struttura dei valori della form. Le chiavi **gender, questionAnswer, secret, username** ed **email** sono i nomi da noi assegnati ai vari campi nella form. **Userdata** invece è un gruppo che abbiamo creato.

* Il secondo invece consente la modifica di un solo valore

this.signupForm.form.patchValue({

userData: {

username: suggestedName

}

});

In questo caso invece si usa la funzione **patchValue** contenuta dentro **form**. Come prima bisogna passare la struttura ma in questo caso non è necessario specificare tutti i valori ma possiamo modificarne solamente un sottogruppo.

**Reactive forms**

Per utilizzare le reactive forms bisogna importare il modulo **ReactiveFormsModule** dentro app.module.

Il metodo reactive prevede la definizione della form lato typescript ed il successivo binding con il template html.

Per prima cosa bisogna dichiarare una variabile di tipo **FormGroup** che conterrà le informazioni riguardo la nostra form. Una volta dichiarata bisogna popolarla con la struttura della nostra form. e.g.:

this.signupForm = new FormGroup({

'userData': new FormGroup({

'username': new FormControl(null, [Validators.required, this.forbiddenNames.bind(this)]),

'email': new FormControl(null, [Validators.required, Validators.email], this.forbiddenEmails)

}),

'gender': new FormControl('male'),

'hobbies': new FormArray([])

});

In questo esempio abbiamo un gruppo chiamato **userData** contr dentro **username** ed **email**, poi abbiamo **gender** ed **hobbies**.

Il gruppo è rappresentato da un ulteriore **FormGroup**, infatti è possibile creare dei **FormGroup** annidati per rappresentare dei gruppi interni di elementi.

Ogni elemento è composto da un **FormControl** che consente di controllare un campo di input (testo, radio, combobox, ...) da controllare.

**FormControl** richiede tre parametri in input di cui solamente il primo obbligatorio:

* Il primo rappresenta il testo di default presente nel campo (negli esempi null).
* Il secondo campo è un array di **validators**. Il validators è la controparte delle directives presenti nelle template forms (come required, email, ...). E’ possibile anche definire dei propri validator.
* Il terzo campo invece è l’array dei **validators asincroni**, cioè i validators che non hanno una risposta immediata ma che richiedono del tempo per la verifica.

Il **FormArray** invece (lo vediamo su hobbies) rappresenta un array di **controller**. Infatti contiene vari **FormControl** che possono essere assegnati dinamicamente.

Per effettuare il collegamento con il template html si utilizzano delle **directives** angular.

Per prima cosa bisogna collegare il nostro **FormGroup** (la rappresentazione della nostra form) con l’elemento form di html.

**<form [formGroup]="signupForm" (ngSubmit)="onSubmit()">**

In questo esempio abbiamo effettuato un property binding collegando la proprietà **formGroup** alla nostra variabile **signupForm** di tipo **FormGroup**, inoltre abbiamo assegnato una funzione all’evento submit della form come nel caso delle **template form** viste precedentemente.

Per collegare i gruppi ed i vari campi invece si utilizzano rispettivamente le directive **formGroupName** e **formControlName**

**<div formGroupName="userData">**

**<input formControlName="username" type="text" id="username" class="form-control">**

Nel primo caso abbiamo una div che ci divide un gruppo di elementi sotto il nome di **userData** mentre nel secondo caso è stata aggiunta una editText con il nome di **username**. E’ possibile vedere nella definizione di esempio vista precedentemente che abbiamo un gruppo **UserData** con dentro un campo **username**. In quel caso questa editText è contenuta all’interno della div.

Come nelle **template form** è possibile ricavare all’interno dell’html informazioni riguardo i vari campi (valid, touched,...). In questo caso però si utilizza la variabile di tipo **FormGroup** che abbiamo definito (nel nostro esempio signupForm). Ad esempio tramite **signupForm.get('userData.username').valid** possiamo capire se il campo **username** contenuto dentro il gruppo **userData** è valido o meno. All’interno della funzione **get** bisogna inserire il path dell’elemento, nel caso in cui non è annidato dentro qualche gruppo bisogna inserire direttamente il nome dell’elemento altrimenti bisogna inserire tutta la path separata da **.** (come nell’esempio).

Per aggiungere altri **controller** all’interno del **FormArray** si procede invece come segue:

**const control = new FormControl(null, Validators.required);**

**(<FormArray>this.signupForm.get('hobbies')).push(control);**

Per prima cosa si definisce il controller poi si effettua il push all’interno dell’elemento effettuandone il caso a **FormArray**. L’elemento è ottenibile tramite funzione **get** e l’identificativo è quello che abbiamo assegnato dentro la definizione della variabile di tipo **FormGroup**. In questo modo è possibile aggiungere un numero qualsiasi di **controller** al nostro template, ovviamente bisogna gestirli lato html. Nel nostro caso vengono gestiti nel seguente modo:

**<div class="form-group" \*ngFor="let hobbyControl of signupForm.get('hobbies').controls; let i = index">**

**<input type="text" class="form-control" [formControlName]="i">**

**</div>**

Viene creata la **div** per ogni **controller** presente all’interno dell’**array**, inoltre come nome al campo di testo viene assegnato l’indice del **ngFor**.

Inoltre è possibile creare dei custom validator tramite:

**forbiddenNames(control: FormControl): { [s: string]: boolean } {**

**if (this.forbiddenUsernames.indexOf(control.value) !== -1) {**

**return { 'nameIsForbidden': true };**

**}**

**return null;**

**}**

Per creare un validator basta definire una funzione che prenda in input un **FormControl** e restituisca un json composto da una lista di coppie stringa: booleano. Queste informazioni verranno poi mostrate nel campo **error** del relativo **FormControl** a cui verrà associato il validator. Il risultato deve essere l’array, nel caso in cui ci siano degli errori mentre **null** nel caso non ci siano (e non false come si potrebbe pensare).

Per assegnare il validator basta aggiungerlo alla lista vista in precedenza (e.g. this.forbiddenNames) senza però passarlo come funzione (cioè senza le parentesi). Inoltre per evitare problemi di scoping è possibile aggiungere un **.bind(this)** in modo da indicare che le variabili utilizzate all’interno del validator sono da ricercare all’interno del nostro typescript (questo perchè, essendo una funzione che viene lanciata da una routine di angular, le variabili con **this.** all’interno del validator non verrebbero cercate dentro il nostro file).

Per ottenere l’informazione che abbiamo generato all’interno dell’html si utilizza invece:

**signupForm.get('userData.username').errors[‘nameIsForbidden’]** -> in questo modo prendiamo il risultato di **nameIsForbidden**. Nel caso di prima avremo **true.**

**Pipes**

Le pipe servono per effettuare delle modifiche all’output nell’html

e.g. {{ var | uppercase }} -> si effettua una string interpolation della variabile **var** e tramite la pipe **|** il contenuto di **var** viene passato ad **uppercase** che trasforma la stringa in maiuscolo.

Le pipe si possono anche contatenare. e.g. {{ varDate | date:’fullDate’ | uppercase }} -> in questo caso la data dentro **varDate** viene passata a **date** che la prepara per mostrarla in modo differente, il risultato di **date** viene passato poi ad **uppercase**. In qualsiasi caso bisogna fare attenzione alle concatenazioni, se mettessimo **uppercase** prima di **date** darebbe errore poichè **varDate** è di tipo **Date** mentre **uppercase** richiede una stringa.

E’ possibile anche passare dei parametri al **pipe**, come nell’esempio di **date** dove passiamo la stringa **‘fullDate’**. I parametri vanno separati da **:** .

E’ possibile anche creare delle pipe custom. e.g.

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

@Pipe({

name: 'shorten'

})

export class ShortenPipe implements PipeTransform {

transform(value: any, limit: number) {

if (value.length > limit) {

return value.substr(0, limit) + '...';

} else {

return value;

}

}

}

In questo esempio è stata creata la pipe **shorten.** All’interno della directive **@Pipe** va inserito il parametro **name** che identifica il nome con il quale la pipe sarà poi chiamata (in questo caso, ad esempio, {{ var | shorten }}.

La classe deve implementare l’interfaccia **PipeTransform** (da importare), questa interfaccia richiede la definizione della funzione **transform**.

I parametri di **transform** sono il valore che prende la pipe ed un numero indefinito di parametri (a discrezione del programmatore). La funzione deve inoltre ritornare un oggetto uguale al valore ottenuto.

**Http**

Per le chiamate http si utilizza il servizio **Http** importato da **@angular/http**.

Per prima cosa si effettua l’inject del servizio nel costruttore. Una volta definita la variabile si possono effettuare le chiamate utilizzando le funzioni della variabile (.**put**, .**post**, .**get**,....).

I parametri di queste funzioni sono: l’indirizzo a cui inviare la richiesta, il body da inviare (se presente) e gli header.

e.g. **this.http.put('https://udemy-ng-http-fa019.firebaseio.com/data', servers, { headers: headers });**

In risposta la funzione restituisce un **Observable**, la richiesta http inoltre viene inviata solamente se viene effettuata una **Subscribe** a tale **Observable** (viene fatto in questo modo per evitare di inviare richieste quando non è necessario -> non serve richiedere cose al server se nessuno vuole ricevere le risposte).

Con l’**Observable** è possibile utilizzare anche la funzione **map** (bisogna importare rxjs-compact).Tale funzione consente di modificare l’output dell’**oservable** in altro mantenendo però il costrutto. In pratica il risultato rimane un **Observable** ma le informazioni vengono prima passate ad una nostra funzione che ci consente di modificarle.

e.g.

this.http.get('https://udemy-ng-http-fa019.firebaseio.com/appName.json').map(

(response: Response) => {

return response.json();

}

);

In questo esempio la risposta dell’**oservable** **get** viene passata dentro la nostra funzione di **map** che non fa altro che estrarre il **body** come **stringa json** e restituirlo. Dall’esterno, quello che otteniamo, è sempre un **observable** che però restituisce una stringa invece che un **Response**.

Oltre a modificare le informazioni possiamo anche catturare gli errori tramite **.catch**.

e.g.

return this.http.get(<https://udemy-ng-http-fa019.firebaseio.com/data>).catch(

(error: Response) => {

return Observable.throw('Something went wrong');

}

);

In questo esempio viene controllato l’errore e restituito come risultato una stringa. Poichè è bene sempre restituire un **Observable** (poichè **get** di default restituisce un **observable** quindi le elaborazioni solitamente riguardano quegli oggetti. Mentre nel caso di catch non viene restituito un **observable** di default) all’interno di **catch** viene costruito e restituito un **Observable** per non rischiare di avere errori.

Inoltre nel caso avessimo una variabile mostrata sull’html tramite string interpolation ed il cui valore dipende da una richiesta lato server(e.g. **get**), possiamo utilizzare la **pipe** **async**. Tale pipe ci consente di non dover effettuare la subscribe alla richiesta ed assegnargli il valore ma si occupa direttamente lei di prendere la response ed assegnarla alla variabile (ovviamente la response che riceviamo deve essere preparata correttamente, magari tramite una map in cui estrapoliamo il body dalla risposta completa, andando ad eliminare i vari header -> come il primo esempio).

**Comandi CLI**

**ng serve** -> Avvia il progetto

**ng new nomeProgetto** -> Crea un nuovo progetto con nome nomeProgetto

**npm install –save bootstrap@3 ->** Aggiunge la libreria css bootstrap al progetto localmente al progetto. Per importare correttamente la libreria all’interno del progetto bisogna:

1. Utilizzare il comando mostrato precedentemente
2. Andare dentro il file angular.json (nella cartella principale) ed aggiungere nel jsonArray “styles” il percorso del modulo di bootstrap. Il modulo è contenuto nella cartella bootstrap dentro node\_modules. (e.g. node\_modules/bootstrap/dist/css/bootstrap.min.css)

**npm install –save rxjs-compat** -> Serve per installare rxjs-compat

**npm install –save firebase** -> Installa firebase sdk

**npm install –save @ngrx/store** -> utilizzato per installare la libreria ngrx per le redux

**ng generate component nomeComponente** (oppure **ng g c nomeComponente**) (opt **–spec false**) **->**  Crea il component nomeComponente in automatico (il parametro opzionale **–spec false** specifica che non bisogna generare il file di testing)

**ng generate directive nomeDirective** (oppure **ng g d nomeDirective**) -> Crea la directive **nomeDirective** automaticamente

**ng generate pipe nomePipe** (oppure **ng g p nomePipe**) -> Crea la pipe **nomePipe** automaticamente

**ng build –prod –aot** (**–base-href /path** opzionale, per cambiare il <base> nel caso in cui l’app non sia al livello di root ma vada inserita in un altro punto) -> Effettua la build dell’app per la produzione e con la funzionalità AoT (ahead of time) che consente di ottenere un eseguibile ancora più stretto omettendo il compilatore e togliendo le funzioni non utilizzate dalle librerie.

**ng add @angular/pwa** -> Utilizzato per aggiungere i service worker