Webpack -> serve per creare dipendenze dei sorgenti/librerie e renderli leggeri

Babel -> transpiler (traduce da un linguaggio sorgente alto livello ad un altro alto livello) per fare transpiling verso vecchie versioni ES5 di javascript

**Angular CLI usa entrambi**

https://update.angular.io/  
ti aiuta per adottare le nuove versioni di angular

npx è la versione di npm che permette di eseguire direttamente un pacchetto

JIT Compilation:  
Il browser scarica il sorgente angular, e lo compila (creando il pacchetto , transpilando i ts in js etc) a runtime .  
  
AOT Compilation:  
Ahead of Time, la compilazione viene eseguita prima di produrre la dist, il browser scarica solo il js finale.  
Molto piu’ performante, ma recente.

Union In Typescript :  
**Serve per vincolare il tipo che puo’ assumere una variabile, o vincolare i valori che puo’ assumere la variabile.  
Es.** quando si fa **let a = string | number -> Stiamo dicendo che la variabile a puo’ essere di tipo string o number.  
Es. function funz(a : number | string) -> stiamo dicendo che l’input puo’ essere solo o un numero o una stringa  
Si puo’ usare non solo per tipi, ma anche per valori veri e propri.  
Es . function ( par : “hello” | “world”) {…} -> stiamo dicendo che deve fallire a runtime se l’input passato alla funzione è un valore diverso da “hello” e “world”**

Type aliases : si usa la keyword **type.**   
Serve per dare un’alias ad un tipo.  
Si puo’ usare sia per dare alias a tipi che siano primitivi/classi es type DictionaryOfStringAndPerson = Dictionary<string, Person>

**stiamo dicendo che DictionaryOfStringAndPerson** è in realtà alias per il tipo definito, e potremo quindi usarlo come tipo delle nostre variabili.  
Si puo’ usare anche in maniera innestata su oggetti  
A computer code with text

Description automatically generated with medium confidence

Ma piu’ frequentemente si usa per dare alias alle nostre union, in maniera tale che se vogliamo definire un vincolo sul valore/tipo di una variabile, che usi union, non dobbiamo ripetere ovunque le union.

Es  
type myCustomType = “value1” | “value2”

Function (par : myCustomType) {…}  
stiamo quindi definendo un type alias, che è una union che blocca su valori stringhe ben definiti e lo stiamo usando per marcare il tipo di input di una funzione, che così andra’ in errore se il valore di input è diverso da value1 e value2  
  
NB: se vuoi usarlo in angular il type alias, devi dichiararlo esportato (con keyword export) come una classe vera e proprio.  
Es  
  
****  
**A screen shot of a computer code

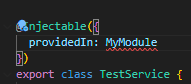
Description automatically generated**

Modularizzazione delle componenti (SENZA STANDALONE)

**NgModule**

Servono per modularizzare il codice.  
E’ buona norma metterli in una cartella apposita, e chiamarli nome.module.ts  
Devono essere classi (esportate con **export** che serve per rendere visibili , es publiche, gli oggetti ts) annotate con @NgModule.  
Nell’@NgModule devono andare  
**\*declarations** : [] la lista delle components/pipes/direttive che quel modulo usa.  
Queste classi devono essere dichiarate con **export**. Questa keywords fa si che il compilatore di typescript (tsc) le veda. In questo modo si dichiarano le componenti per il rendering nel modulo.  
NB: di default, chi importa un modulo, non ha accesso alle componenti che stanno nelle sue declaration (cioè che lui usa).  
Quindi se un modulo A, importa un modulo B , e vogliamo che A possa usare le componenti che stanno nelle declaration di B abbiamo 2 strade:  
-si ridichiarano anche nelle declaration del modulo A (sporco, poiché si duplica la presenza nelle declaration)  
-si dichiarano sempre nel modulo B, oltre che nelle declaration, anche negli **exports:   
\*gli exports di un module sono le componenti (pipes, components, direttive) che saranno utilizzabili dagli altri moduli che importano il modulo che le ha dichiarate negli exports.**  
 **\*imports:** è dove si dichiarano gli altri moduli di cui ha bisogno il modulo che stiamo dichiarando.  
Quando si inserisce un modulo B negli imports del modulo A, il modulo A avrà accesso a :  
-tutte le componenti/pipes/direttive che il moduloB ha oltre che nelle declaration anche negli exports.  
-ai servizi di cui il moduloB è provider (poiché inseriti nei providers dichiarati in moduloB o perché nei servizi c’e’ l’@Injectable col providedIn: moduloB)  
  
 **\*providers:**  è dove si dichiarano i services di cui questo modulo è provider. Un service è un iniettabile (depedency injection) perché c’e’ un provider a lui associato. In genere quando creiamo un service con angular cli facendo **ng g service**  automaticamente questo viene annotato con

A screen shot of a computer program

Description automatically generated  
Questo vuol dire che il provider associato al service (affinchè possa essere iniettato dalla dependency injection nei costruttori) è quello di default, il root provider. Il root provider fa si che un elemento possa essere iniettato nei costruttori di qualunque parte del codice dell’applicazione.  
Se invece non vogliamo usare il root provider, ma vogliamo che un servizio sia iniettabile soltanto da un modulo B che negli imports ha importato un modulo A, allora come provider associato al service dobbiamo togliere il root , e dargli come provider quello del modulo.  
Abbiamo 2 modi per farlo.  
  
Sul service stesso :  
Es   
  
In questo modo soltanto i moduli che importano (negli imports) MyModule possono avere l’injection del TestService.

Il secondo modo invece è non farlo con @Injectable sul service stesso (quindi si cancella @Injectable sul service) e si usa il campo **providers** di MyModule   
Quindi se il mio modulo MyModule nell’annotazione NgModule ha tra i providers TestService

A screen shot of a computer program

Description automatically generated

Stiamo dicendo che il modulo in questione sarà il provider per TestService.  
E quindi i moduli che vorranno avere nei loro componenti l’injection di testservice abilitata, dovranno fare l’import di MyModule  
  
**NB: gli imports sono transitivi tra moduli.  
Es se moduloA importa moduloB e moduloB importa moduloC, e moduloC è provider di un servizio, allora tale servizio sarà visibile sia in moduloB che in moduloA.  
Stessa cosa per le declaration**

**NB: un modulo per esportare una SUA componente deve per forza averla nelle declaration (quindi se sta negli exports deve stare nelle declaration) viceversa se la ha nelle declaration non deve per forza averla negli exports.  
NB2: un modulo puo’ utilizzare una componente che formalmente appartiene ad un altro modulo, senza importare quel modulo ,ma mettendola direttamente nelle sue declaration, ma questa e’ una pessima pratica, poiché ha 2 problemi:**

**funziona solo se il modulo che mette nelle declaration direttamente la componente di un altro modulo, non importa anche il modulo. Se importa anche il modulo, allora non compila (poiché avremo una componente che compare nelle declaration di 2 moduli non indipendenti perché uno importa l’altro)  
Molto meglio invece fare l’import del modulo a cui appartiene formalmente la componente (presente anche negli exports di tale modulo). In questo modo ,nel modulo che vorrà usarla non bisogna fare nient’altro (solo l’import dell’altro modulo) per poter usare la componente esportata dall’altro.  
  
Si nota come pero’ questo complica la vita, poiché ogni volta che si vuole usare una componente, pipe o direttiva custom , bisogna creare un modulo associato che le esporti.  
Ecco perché il seguito.**

NB: il modo in cui fino ad ora è stata spiegata la modularizzazione delle dipendenze in angular (moduli, componenti, pipes, direttive, imports, exports, providers etc) riguarda il modo classico pre ANGULAR 14. Da dopo Angular 14 , per semplificare la vita, sono nati gli standalone.  
Uno standalone (che sia una componente, una direttiva, o una pipe, MA NON UN SERVICE) puo’ essere tutto quello che PRE angular 14 doveva dipendere da un modulo per essere usata .  
  
Per generare da CLI elementi standalone si aggiunge –standalone alla fine (es ng g c – standalone).  
Un elemento standalone semplicemente ha il flag standalone : true nel decoratore che lo dichiara

(es @Pipe, @Component etc).  
  
Un componente/pipe/direttiva standalone quindi non ha un modulo associato.  
Ma come lo si usa quindi ?  
  
Se un elemento **NON STANDALONE** vuole usare uno **standalone**, allora nel modulo di questo elemento NON standalone si aggiunge agli imports l’elemento standalone (PROPRIO COME SE SI STESSE IMPORTANDO UN MODULO, ma un modulo non è, quindi da angular 14 negli imports di un @NgModule non compariranno piu’ solo moduli! ).  
(nell’esempio visto che nel componente AppComponent, che non è standalone, vogliamo usare BComponent, che è standalone, allora nel modulo di AppComponent negli imports inseriamo BComponent)  
A black screen with white text

Description automatically generated

Se un elemento **STANDALONE** vuole usare un altro elemento **STANDALONE** si utilizza un campo imports presente proprio nel decoratore dell’elemento (quindi gli imports non sono piu’ ad uso esclusivo dei moduli ma anche delle componenti/pipe/direttive standalone stesse, quindi ad esempio troveremo imports : [] anche nei @Component, @Pipe etc).  
(nell’esempio BComponent, che è una componente standalone quindi senza modulo, sta importando una pipe che è anch’essa un elemento standalone, quindi senza un modulo associato)

A screen shot of a computer code

Description automatically generated  
  
  
Se un elemento **STANDALONE** vuole usare uno **NON STANDALONE**, per definizione quello NON STANDALONE ha un modulo associato (chiamiamolo moduloB).  
Quindi se la componente A vuole usare componente B che è associata a modulo B, nella componente A avremo un import : in cui compare moduloB.  
(nell’esempio BComponent è standalone, e vuole usare nel suo template <app-a> cioè AComponent, che non è standalone, quindi BCOmponent dovrà importare il modulo di A, AModule, dove c’e’ l’export di AComponent>

A screen shot of a computer code

Description automatically generated

**Quindi in definitiva se un elemento (component, pipe o direttiva) non è standalone, se qualcun altro vuole usarlo deve importare il suo modulo, e se l’elemento vuole usare roba di altri , gli import devono stare nel suo modulo.  
Se invece un elemento è standalone, si fa lui direttamente gli import di quello che gli serve, e chi vuole usarla importa lui direttamente .**

**A seconda della combinazione di cosa è chi importa e cosa è chi è importato ci sono le combinazioni:  
non standalone importa non standalone -> il modulo importa il modulo  
non standalone importa standalone -> il modulo importa componente  
standalone importa non standalone -> la componente importa il modulo  
standalone importa standalone -> la componente importa la componente**

**In typescript il cast è detto type assertion e si fa con   
let a = (obj as string)**

**Nella cartella /assets tutto quello che viene messo viene copiato pari pari nella build (quindi si mettono immagini, risorse etc)**

**Interpolation -> {{ variabile }}**

**Property binding -> assegnazione del valore di un espressione ad una proprietà  
es <img [url]=”variabile” />**

**NB: la differenza è che il property binding puo’ essere usato anche per espressioni che valutano non a stringhe, mentre l’interpolation traduce sempre a stringhe**

**Event binding -> ( evento ) = “funzione()”**

**Come passare dati tra componenti:**

**da padre a figlio -> il padre fa property binding su una variabile del figlio.  
La variabile nel figlio deve essere dichiarata con @Input**

**Es  
Padre  
A screen shot of a computer code

Description automatically generated  
Figlio  
A screen shot of a computer program

Description automatically generated**

**da figlio a padre -> si usa event emitter.  
Il figlio ha un event emitter parametrizzato al tipo di dato che deve viaggiare, e annotato con @Output()  
Es  
  
Quando il figlio vuole mandare dati, chiama il .next(dato) sull’event emitter , trattandolo come un observer.  
Il padre registra sull’event emitter del figlio che viene trattato come un evento a tutti gli effetti un handler, usando la notazione ( )   
Es   
A black background with white text

Description automatically generated  
E si registra l’handler che deve prendere in input $event, che viene valorizzato col dato che sta viaggiando.**

**In typescript se si vogliono incapsulare delle variabili, rendendole private e accessibili solo con getter e setter (poiché magari la variabile va modificata prima di settarla) esiste una sintassi specifica diversa:  
  
il getter ed il setter saranno funzioni che si chiamano come una variabile, e la variabile reale è buona norma chiamarla diversamente.  
Es  
  
A screen shot of a computer code

Description automatically generated  
  
E questo vale per ogni volta che vogliamo dichiarare una finta variabile di classe, publica, si scrive  
get nomevariabile() {…} e questo fa si che si puo’ chiamare la variabile di componente ma in realtà sta venendo chiamato il metodo nomevariabile() , e stessa cosa per il set quando lo vogliamo settare e questo ci permette di avere differenti variabili che in realtà sono proxy verso quella reale.**

**Questo quando si puo’ usare ?  
Un usecase è quando vogliamo avere un binding su una property di un componente, ma vogliamo che i valori in input vengano trattati prima di settarli.  
Invece di avere l’annotazione @Input sulla variabile, si mette sui getter e setter**

**A computer screen with text

Description automatically generated**

Content Projection  
Quando una componente padre fa injection di html in un figlio.  
Il padre scrive il codice html all’interno dei tag della direttiva del figlio.  
Il figlio dichiara un <ng-content></ng-content> laddove vorrà mostrare il contenuto injected dal padre.  
Es  
Padre  
A screen shot of a computer program

Description automatically generated

Figlio  
A screen shot of a computer

Description automatically generated

Per applicare uno stile al template selector, nel foglio di stile associato al template si puo’ usare

**:host**Se invece vogliamo applicare un css sull’host (cioè il template selector) ma soltanto se un suo antenato ha una condizione di classe abilitata si usa

**:host-context(.classe da valutare)**

**Es**

**:host-context(.active) applica la regola css all’intero component se uno dei suoi antenati ha la classe active su di esso**

**NB: quando si dice CONTENT ci si riferisce a quello che è passato come content projection, cioè passato ad una direttiva figlia che verrà mostrato nel suo <ng-content>  
quando si dice VIEW ci si riferisce al template di un component (l’html associato)**

**Lifecycle hooks**

**(il nome del metodo è uguale a ng[NomeInterfaccia])**

**Costruttore componente -> chiamato per primo**

**ngOnChanges -> chiamata se e solo se il componente ha binding su input, ed è chiamata in tal caso all’inizio PRIMA dell’ngOnInit e dopo ogni cambiamento**

**ngOnInit -> chiamata all’inizio quando il componente è inizializzato (dopo che sono state settati i bindings)**

**ngAfterContentInit -> chiamata dopo che è stata risolta la content projection del componente in cui viene iniettata con <ng-content>**

**ngAfterContentChecked -> chiamata dopo che è terminata la ngAfterContentInit, e dopo ogni cambiamento**

**ngAfterViewInit -> chiamata dopo che è stata risolta e inizializzata la view (il template html) del componente**

**ngAfterViewChecked -> chiamata dopo la ngAfterViewInit, e dopo ogni cambiamento**

**NB: ngAfterContentInit e ngAfterContentChecked vengono sempre chiamate prima di ngAfterView Init e ngAfterViewChecked perché angular prima risolve i content projection (negli <ng-content>) e poi risolve la view.  
NB: vengono sempre risolte le componenti dal basso verso l’alto, quindi quando parte la ngAfterViewInit di una componente padre, sicuramente ci saranno già i figli valorizzati (che quindi potremo prenderci con @ViewChild)**

**Dynamic component rendering**

Tipicamente un template (view) associato ad un componente angular è del codice html statico, con delle variabili dinamiche, e con all’interno delle componenti figlie.  
Quali siano le componenti figlie è definito in maniera statica nella dichiarazione dell’html.  
Ma come fare se vogliamo avere un template view di una componente padre, che carica a runtime e decide (a seconda di determinate condizioni) quale componente figlia mostrare ?   
Potremmo fare la cosa usando \*ngIf in parallelo che si attivano o disattivano mostrando la componente associata alla condzione, ma se queste sono tante diventa poco manutenibile.  
Un’alternativa è usare la direttiva \***ngComponentOutlet  
La direttiva \*ngComponentOutlet ci permette di caricare a runtime un component, e di caricare anche i valori che dovranno essere usati per iniettare nei property binding della componente figlia caricata a runtime.  
  
Es immaginiamo di avere l’app-component che a seconda del bottone premuto carica come componente figlia app-b o app-a  
Si usa un <ng-container su cui si mette la direttiva \*ngComponentOutlet che si aspetta 2 parametri, il primo il componente da renderizzare (includere) e il secondo dopo la stringa input: gli inputs da usare per il databinding che dovranno avere la forma di un’oggetto js con le property chiamate come gli input di cui fare binding.  
NB: l’input binding così funziona solo da angular 16 , prima il binding era leggermente diverso.  
  
Quindi nel nostro esempio avremo le 2 componenti figlie, ognuna con un semplice da binding in ingresso (variabile message),   
avremo poi un service iniettato nella componente padre app-a che a seconda del type selezionato ritorna una componente, e una mappa di input, e poi avremo la componente app-a padre che permette di selezionare un type o un altro, e a seconda di questo ritorna il componente.  
  
Quindi le 2 componenti figlie con dentro un semplice binding :  
A screen shot of a computer program

Description automatically generated**

**A screen shot of a computer program

Description automatically generated**

**Il component provider che a seconda del tipo ritorna la componente (questo viene iniettato nel componente padre app-a)  
  
A screen shot of a computer program

Description automatically generated**

**questa è la componente padre  
  
A computer screen shot of a program code

Description automatically generated**

**E la view associata al padre dove c’e’ la direttiva ngContentOutlet  
(notare che selectedComponent anche se nel codice ts è un metodo, viene trattato come variabile poiché è un getter, e questo ci è utile in quanto permette di aggiungere il codice che fa da proxy alla chiamata col selected type al servizio)  
  
A screen shot of a computer code

Description automatically generated**

Angular dev tools è un estensione che permette nella console di vedere nativamente gli oggetti (componenti, input binding , gerarchie etc) di un app scritta in angular.  
Funziona solo su chrome

Pipes

Le pipes servono per trasformare dei valori.  
Dal punto di vista delle dichiarazioni, sono trattate come le componenti.  
Un modulo che le usa le ha nelle declaration, e se le esporta verso altri moduli che le importano le avrà negli exports.  
Di default angular mette a disposizione nel common module un sacco di pipe predefinite.  
Per vedere quali sono, come funzionano, e quale modulo le esporta, si va nell’angular documentation sotto **reference >API reference e si cercano le pipe.**

NB: la convenzione è che una pipe si chiama NomePipe e viene usata con | nome  
Ad esempio la **DatePipe** viene usata con {{ variabileData | date }} e questo è l’uso di default.  
Se vogliamo usare una formattazione differente, dobbiamo passare il date format in input alla pipe.  
Gli input alla pipe si passano con la sintassi nomepipe : input  
Quindi ed esempio  
{{ variabileData | date : “d/MM/yyyy”}  
sempre nell’api reference se cerchiamo per la DatePipe ci sono sia i formati custom (da passare in input come stringa es dd/MM/yyyy sia i predefined dove si passa in input la stringa che rappresenta la formattazione predefinita es “short” )

Un’altra pipe di default interessante è la **JsonPipe, quindi usata con | json**  che permette di printare il json di un oggetto ts/js.

**SE VOGLIAMO DICHIARARE UNA NOSTRA PIPE  
Dichiariamo una classe (ovviamente pubblica quindi con export) che implementa l’interfaccia PipeTransform, ed è annotata con @Pipe ({name: “nomepipe”})**

**Es**

**A screen shot of a computer program

Description automatically generated**

**Implementiamo il metodo dell’interfaccia PipeTransform, transform che prende in input come primo argomento il valore che stiamo trasformando nella pipe, e come opzionale la lista degli input alla pipe (gli input alla pipe ricorda sono gli elementi che possono indicare come deve funzionare la pipe ad esempio nella DatePipe sono il formato di data da mostrare)**

**A questo punto la pipe va nelle declaration del modulo che le usa.   
Se il modulo vuole rendere la pipe disponibile anche alle componenti di altri moduli, dovrà metterlo anche negli exports (proprio come per le componenti).  
Quindi se altri moduli vogliono usarla basterà che importino il modulo che la ha negli exports.  
Se per esempio la pipe è dichiarata nel modulo A  
A screen shot of a computer program

Description automatically generated  
(notare come stia sia nelle declarations, sia negli exports..se questa pipe doveva essere disponibile solo per le components dell’AModule bastava metterlo nelle declaration, ma noi vogliamo che sia disponibile anche per componenti di altri moduli quindi l’abbiamo messa negli exports)**

**A questo punto se vogliamo usarla in un altro modulo, nel modulo App, basta che quest’altro modulo metta negli imports l’AModule  
  
A screen shot of a computer program

Description automatically generated**

**E per usarla nelle sue componenti**

**A screen shot of a computer

Description automatically generated  
  
selectedType sarà il value passando nel transform della pipe, e “salutami” sara’ il primo elemento degli args.**

Direttive

Ci sono 2 tipi di direttive:

Direttive di Attributo-> cambiano il comportamento/stile visivo di un componente  
Direttive Strutturali -> cambiano la struttura/layout di un componente

**NB: le direttive, come le componenti e le pipelines, sono dichiarate come classi (publiche quindi con export) , poi sono presenti tra le declarations del modulo che le vuole usare, e se questo modulo vuole renderle usabili dalle componenti di altri moduli le avrà negli exports.**   
Le direttive possono essere quelle built-in o dichiarate custom da noi.  
Le direttive built-in di tipo **ATTRIBUTE** classiche sono:

**NgClass (esportata dal modulo Common) serve per aggiungere classi**

****

**ngStyle(esportato dal modulo Common) serve per aggiungere stili**

**  
e nel componente**

**(nell’esempio di qui sto usando la tecnica del getter per fare proxy sulla variabile che contiene gli stili)  
A screen shot of a computer program

Description automatically generated**

**NB: i record servono per bloccare il tipo, rappresentando un oggetto javascript chiave valore, il primo generic indica il tipo della chiave, il secondo del value.**

Ng-container vs ng-template  
ng-container -> si usa quando serve un elemento per metterci una direttiva, visto che angular non accetta 2 direttive strutturali sullo stesso dom (es \*ngIf e \*ngFor) ma non vogliamo aggiungere ad esempio un div. Si crea un ngContainer su cui si piazza la direttiva.  
l’<ng-container> non diventa un dom !!!  
  
ng-template invece crea un dom di default nascosto, e si usa quando si vuole creare un dom di fallback ad esempio. Tale dom viene generato e mostrato solo quando esplicitamente si va nel fallback.  
Es  
  
<div \*ngIf=”condizione; else fallback” >…

</div>

<ng-template #fallback>..

</ng-template>  
  
Se fallisce l’if viene mostrato il contemuto dell’ng-template.