TỰ HỌC ANGULAR

Angular là 1 framework dung để build SPA

Main.ts : 1 ứng dụng Angular thì file main.ts là file đầu tiên nó được chạy khi khởi động dự án Angular. Đây là 1 ts module, là thành phần của một tập hợp các module khác nhau, trong đây có import các module khác từ thư viện bên ngoài, từ file local.

App.module.ts: Bản thân là 1 typescript module, có thể import từ bên ngoài vào để sử dụng hoặc export ra để các file khác sử dụng.

## **- Decorator:**

@NgModule: Typescript Decorator : là cú pháp liên quan đến typescript chứ không phải của riêng Angular nhưng Angular sử dụng Decorator để cung cấp Metadata

+, Bên trong Metadata:

* declarations: tên lớp view thuộc về module này
* exports: danh sách tên các module hoặc component có thể sử dụng module này
* imports: tên các module sẽ được dùng từ module này
* providers: tên các service sẽ được dùng từ module này, chúng ta sẽ tìm hiểu về service sau
* bootstrap: tên lớp view dành cho root module, chỉ có root module mới thiết lập tham số này

- Nguyên tắc tạo 1 component:

Tạo 1 file mới, tạo 1 class trong file đó, declare class đó bằng component, khai báo component mới này trong AppModule

**-Properties binding:** để hiển thị dữ liệu

Cú pháp : [type] = “‘text’” or [type] = “inputType”, biến inputType sẽ được định nghĩa trong AppComponent

Bind các biến trong class component ra template gọi là property binding

[ngStyle]=“{color: colorText, fontSize: textLarge}”

[ngClass] =“{“isActive”: isActiveNav}” => true : có class, false: không có class

**- Event binding** : nhận sự kiện từ template và xử lý trong app component

Cú pháp : (click) = “handler()”, hàm handler() sẽ được định nghĩa trong AppComponent

Khi người dung tương tác với template: click, scroll,… đó gọi là event => event đó được component listen và handle => gọi là event binding

**- Kết hợp giữa Properties binding và Event binding gọi là Two-way data binding**

Cú pháp : [(ngModel)]=”name”

Tương đương : [ngModel] = “name” (ngModelChange) = “name = $event”

- Structure Directive là một đoạn Directive có thể thay đổi cấu trúc của DOM, có thể lấy đi, thêm vào cấu trúc của DOM

Trong Angular để thêm, xóa, thay đổi structure (structure HTML chẳng hạn) ở trên view của component chúng ta sẽ dùng Structure Directive.

- Life cycle hook:

Là một hàm, một method tự động chạy dựa trên một sự kiện nào đó xảy ra trên component. VD như component được khởi tạo,…

|  |  |
| --- | --- |
| **Hook** | **Mục đích & thời điểm** |
| ngOnChanges() | Sinh ra khi Angular thiết lập các thuộc tính đầu vào để ràng buộc dữ liệu. Phương thức nhận một đối tượng SimpleChanges của các giá trị thuộc tính hiện tại và trước đó. Được gọi trước ngOnInit() và bất cứ khi nào một hoặc nhiều thuộc tính đầu vào ràng buộc dữ liệu thay đổi. |
| ngOnInit() | Khởi tạo các directive/component sau khi Angular hiển thị các ràng buộc dữ liệu và thiết lập các thuộc tính đầu vào của directive/component’s(@Input() )  Được gọi duy nhất một lần sau ngOnChanges(). |
| ngDoCheck() | Phát hiện và hành động theo những thay đổi mà Angular  không thể hoặc sẽ không tự mình phát hiện. Được gọi trong  mỗi lần chạy phát hiện thay đổi, ngay sau ngOnChanges() và  ngOnInit(). |
| [ngAfterContentInit()](https://angular.io/api/router/RouterLinkActive) | Thực thi sau khi Angular thêm nội dung bên ngoài vào view của component / view mà directive được đưa vào. Được gọi một lần sau ngDoCheck() đầu tiên. |
| ngAfterContentChecked() | Thực thi sau khi Angular đã kiểm tra nội dung bên ngoài đã được đưa vào view của component. Được gọi sau ngAfterContentInit() và mọi ngDoCheck() tiếp theo. |
| [ngAfterViewInit()](https://angular.io/api/forms/NgForm) | Thực thi sau khi Angular khởi tạo các view của component và các view con / view mà directive được đưa vào. Được gọi một lần sau ngAfterContentChecked() đầu tiên. |
| ngAfterViewChecked() | Thực thi sau khi Angular kiểm tra các view của component và các view con /view mà directive được đưa vào. Được gọi sau ngAfterViewInit() và mọi ngAfterContentChecked() tiếp theo. |
| ngOnDestroy() | Dọn dẹp ngay trước khi Angular phá hủy directive / component. Hủy đăng ký Observables và tách trình xử lý sự kiện để tránh rò rỉ bộ nhớ. Được gọi ngay trước khi Angular phá hủy directive / component. |

## **- ngIf:**

**CẤU TRÚC IF-ELSE**

Để hiển thị một phần view (template) theo một điều kiện, chúng ta sẽ gắn thêm một property đặc biệt vào một tag, với cú pháp có chứa dấu \* (asterisk) như sau **\*ngIf="expression"**:

```

@Component({

selector: 'app-hello',

template: `

<h2>Hello there!</h2>

<h3>Your name: {{ user.name }}</h3>

<p>Your name: {{ user.age }}</p>

<div \*ngIf="user.age >= 13">

Bạn có thể xem nội dung PG-13

</div>

`

})

export class HelloComponent {

user= {

name: 'Tiep Phan',

age: 30

};

}

```

Chỉ cần có thế là chúng ta có thể hiển thị view tùy thuộc vào dữ liệu mà expression trả về. Truthy thì hiển thị, Falsy thì không hiển thị.

Với những directive được cung cấp sẵn (built-in) bởi Angular, giờ đây HTML template của component sẽ rất flexible.

Vậy nếu chúng ta muốn dùng IF-ELSE thì thế nào. Có thể các bạn sẽ nghĩ ngay đến phủ định mệnh đề IF là được ELSE thôi. Điều này hoàn toàn được.

```

<div \*ngIf="user.age >= 13">

Bạn có thể xem nội dung PG-13

</div>

<div \*ngIf="user.age < 13">

Bạn không thể xem nội dung PG-13

</div>

```

Hoặc chúng ta có cách hay ho khác, đó là dùng đến **ng-template**. Tag **ng-template** là một tag được cung cấp bởi Angular, nó sẽ lưu trữ một Template được định nghĩa bên trong cặp thẻ mở/đóng của nó. Những gì được định nghĩa bên trong đó sẽ không được hiển thị ra view, nhưng chúng ta có thể sử dụng Template đó để render bằng code. Đoạn code phía trên có thể được chuyển đổi tương đương:

```

<div \*ngIf="user.age >= 13; else noPG13">

Bạn có thể xem nội dung PG-13

</div>

<ng-template [#noPG13](https://www.facebook.com/hashtag/nopg13?__eep__=6&__gid__=184402672058592&__cft__%5b0%5d=AZUCmXQwIHlPGKsAUGeC00iIHeAi4p8_oLu3Gn1PV1ASG69C61Ychp-UOAfCfGjMlk8ESDwMMUJ_k37RX3YYqJ1dPhh8j-GqI07ZOmcoFN-Oo85p7B0g5qXG8H0ax6rM8LjxcbAqv5TWfuK_dEes66Ad&__tn__=*NK-R)>

<div>

Bạn không thể xem nội dung PG-13

</div>

</ng-template>

```

**NG-TEMPLATE**

Với cú pháp sử dụng dấu \* ở trên, có thể các bạn sẽ thấy nó khác lạ, nhưng thực tế, nó được gọi là **Syntactic sugar** (giúp nhìn code dễ hiểu, dễ đọc hơn chẳng hạn) được chuyển đổi sang dạng property binding như sau:

```

<ng-template [ngIf]="user.age >= 13" [ngIfElse]="noPG13">

<div>

Bạn có thể xem nội dung PG-13

</div>

</ng-template>

```

## **- ngFor Directive:**

Dùng để lặp qua một danh sách để hiển thị danh sách này trên HTML template.

Dấu hoa thị \* trước một Directive gọi là Structure Directive

Cú pháp : <li let user of users></li> tương tự như for (let user of users){}

**MỘT SỐ LOCAL VARIABLE TRONG MỘT NGFOROF TEMPLATE**

Khi sử dụng NgForOf, ở mỗi vòng lặp chúng ta sẽ có thể truy xuất đến một số local variable như:

* $implicit: T: Giá trị của phần tử trong danh sách ở lần lặp hiện tại
* index: number: index của lần lặp hiện tại
* count: number: số lượng phần tử trong danh sách
* first: boolean: True nếu đây là phần tử đầu tiên trong danh sách
* last: boolean: True nếu đây là phần tử cuối cùng trong danh sách
* even: boolean: True nếu đây là phần tử ở index chẵn
* odd: boolean: True nếu đây là phần tử ở index lẻ

Để truy xuất được những biến trên chúng ta chỉ cần gọi chúng như sau:

Đối với $implicit nó sẽ được gắn cho biến mà chính chúng ta khi khai báo let something of xxx, lúc này something = $implicit.

Đối với các biến khác, chúng ta chỉ cần dùng cú pháp sau:

```

<div \*ngFor="let author of authors; index as idx; count as total">

({{idx}})/({{total}}): {{author.id}} - {{author.firstName}} {{author.lastName}}

</div>

```

Lúc này biến idx = index, và total = count. Tương tự đối với các biến còn lại.

**CẤU TRÚC NGFOROF VÀ NG-TEMPLATE**

Với cấu trúc dùng dấu sao (\*) như trên chúng ta có thể chuyển đổi tương ứng về dạng ng-template và property binding như sau:

```

<ng-template ngFor [ngForOf]="authors" let-author let-idx="index" let-total="count">

<div>

({{idx}})/({{total}}): {{author.id}} - {{author.firstName}} {{author.lastName}}

</div>

</ng-template>

```

**- ATTRIBUTE DIRECTIVE**

Attribute directive là gì, nó có khác gì Structure directive mà chúng ta đã gặp qua như NgIf, NgForOf không?

Đối với Structure directive, nó có thể thêm, xóa, thay đổi cấu trúc cây DOM. Còn một Attribute directive sẽ có thể thay đổi cách hiển thị (style) hoặc hành vi của một DOM element/Component/Directive khác. Đấy chính là sự khác biệt giữa chúng.

## **- Class Binding:**

Trong các ứng dụng thực tế, có thể chúng ta cần thay đổi (thêm, xóa) một số class tùy thuộc vào một số điều kiện nào đó.

Ví dụ, nếu chúng ta đang chọn một tab nào đó để hiển thị, thì tab đó sẽ có thêm class tab-active, các tab khác sẽ không có. Lúc này chúng ta sẽ sử dụng cú pháp:

<div [class.tab-active]="isTabActive">

some content

</div>

Nhìn qua thì nó chỉ là property binding, với giá trị của isTabActive trả về true thì classList của div đó sẽ tồn tại class tab-active, còn nếu trả về false thì sẽ không tồn tại.

Ngoài cú pháp trên chúng ta có thể dùng:

[class]="classExpr"

Với classExpr có thể là string, array string hoặc object – nếu key nào của object là truthy thì sẽ thêm vào, nếu falsy thì sẽ xóa đi.

Ví dụ các dạng của classExpr:

String: "my-class-1 my-class-2 my-class-3"

Array String: ['foo', 'bar']

Object: {foo: true, bar: false}

Tương tự với class binding chúng ta có thể sử dụng ngClass, nhưng hiện tại cũng có thể nói rằng ngClass không có gì khác biệt với [class]=”classExpr”. Và cách sử dụng Class Binding vẫn được khuyến cao sử dụng hơn ngClass.

=>

- [class.ten-class]

- [class] = “array”

- [class]= “object” { [ten-class: string]: boolean }

- [class]= “variable”

## **- Style Binding:**

Có thể khi cần thiết, chúng ta cần binding cho style property (inline style), lúc này chúng ta có thể sử dụng Style binding.

Cấu trúc của style binding như sau:

[style.property]=”expression”

Với expression sẽ tính toán về các giá trị kiểu string | undefined | null

Ví dụ:

<div [style.width]=”someValue”></div>

Tiếp theo là cú pháp kèm theo unit:

[style.property.unit]=”expression”

Với expression sẽ tính toán về các giá trị kiểu number | undefined | null

Ví dụ:

[style.height.%]=”containerHeight”

Cuối cùng là cú pháp dạng:

[style]=”styleExpr”

Với styleExpr là một trong các dạng:

String: "width: 100%; height: 100%"

Array String: ['width', '100px']

Object: {[key: string]: string | undefined | null} như {width: '100px', height: '100px'}

Có một directive tương tự là ngStyle với cách dùng gấn giống thế, trong hầu hết các trường hợp, chúng ta được khuyến cáo sử dụng style binding thay thế.

Lưu ý rằng, một style property có thể dùng cả kiểu dash-key hoặc camelCase, ví dụ font-size hoặc fontSize đều được.

=>

- [style.tenStyle]

- [style.ten-style]

- [style.tenStyle.unit

- [style]= “object”

- [style.css-variable]

## **- @Input Decorator**

Để thêm một property cho phép thiết lập progress hiện tại của thanh progress-bar, chúng ta có thể khai báo một property cho class và thêm một móc móc (decorator) như sau:

export class ProgressBarComponent implements OnInit {

@Input() progress = 0;

constructor() { }

ngOnInit() {

}

}

Đơn giản thế là chúng ta đã có thể báo cho Angular biết rằng, chúng ta cần component này nhận vào một property tên là progress, và giá trị mặc định của nó bằng 0.

Bây giờ, chúng ta có thể thêm một số property khác cho component như property về màu sắc chẳng hạn.

export class ProgressBarComponent implements OnInit {

@Input() backgroundColor: string;

@Input() progressColor: string;

@Input() progress = 0;

constructor() { }

ngOnInit() {

}

}

Như vậy, component của chúng ta sẽ có thể nhận vào 3 properties, với property progress sẽ có thể chứa giá trị default bằng 0.

Lưu ý rằng, @Input được gọi là một property decorator, nó sẽ gắn thêm meta data cho property ngay phía sau nó.

Nếu bạn không khai báo decorator @Input thì sẽ không thể nhận giá trị truyền vào từ component khác, vì Angular sẽ không biết cách để binding, và do đó property của bạn chỉ là một property bình thường của class.

Khi đã có component và khai báo input rồi thì làm sao để sử dụng. Các bạn cần nhớ lại property binding ở trong những buổi đầu tiên, lúc này chỉ cần dùng cú pháp vuông vuông để binding cho property các bạn muốn là được rồi.

<app-progress-bar

[progress]="15"

[backgroundColor]="'#9e9e9e'"

[progressColor]="'#2e8b57'"

></app-progress-bar>

Đây là component progress-bar của chúng ta.

import { Component, OnInit, Input } from '@angular/core';

@Component({

selector: 'app-progress-bar',

template: `

<div class="progress-bar-container" [style.backgroundColor]="backgroundColor">

<div class="progress" [style]="{

backgroundColor: progressColor,

width: progress + '%'

}">

</div>

</div>

`,

styles: [`

.progress-bar-container,

.progress {

height: 20px;

}

.progress-bar-container {

width: 100%;

}

`]

})

export class ProgressBarComponent implements OnInit {

@Input() backgroundColor: string;

@Input() progressColor: string;

@Input() progress = 0;

constructor() { }

ngOnInit() {

}

}

##### NGONINIT VS CONSTRUCTOR

Khi làm việc các bạn có thể thấy rằng Angular giới thiệu một số method gọi là life-cycle, chúng làm nhiệm vụ gì.

Theo như lẽ thông thường, con người từ khi sinh ra đến khi chết đi sẽ đều trải qua một số sự kiện trọng đại trong đời, ví du: sinh ra, đầy tháng, sinh nhật hàng năm, kết hôn, chết đi. Tương tự như thế trong ứng dụng Angular sẽ có vòng đời cho các component. Angular được xây dựng xung quanh Component và Directive, ở một thời điểm có thể có 1 component được khởi tạo, một thời điểm khác chúng lại được xóa đi khỏi view, và có một số “sự kiện” khác cũng được xảy ra, do đó Angular cung cấp cho chúng ta một số method mà chúng ta chỉ việc khai báo cho component/directive, còn lại Angular sẽ tự call khi có những sự kiện tương ứng xảy ra.

Ở trong code phía trên, chúng ta được giới thiệu constructor và ngOnInit, vậy chúng giống và khác nhau gì?

Constructor là hàm tạo của một class, nó là một function đặc biệt mà khi bạn khởi tạo một instance của class thì nó sẽ được tự động chạy, và chỉ chạy duy nhất một lần.

ngOninit là một life-cycle method, nó sẽ được Angular tự động gọi khi component được khởi tạo, sau khi constructor chạy và sau khi các input đã được binding.

Do đó nếu bạn binding cho một property ở template của component cha, thì ở constructor của component con bạn sẽ chưa nhận được giá trị mà bạn đã binding, nhưng ở ngOnInit thì bạn sẽ có thể nhận được.

Trong thực tế, Angular khuyến cáo hạn chế code ở constructor, constructor làm càng ít nhiệm vụ càng tôt, hãy để ngOnInit lo tiếp phần việc còn lại.

##### THAY ĐỔI GIÁ TRỊ CỦA INPUT.

Giả sử trong trường hợp nào đó mà chúng ta không biết người dùng binding những dữ liệu có hợp lệ không (điều này hoàn toàn có thể xảy ra như trường hợp người dùng truyền data kiểu any vào chẳng hạn), và chúng ta muốn validate Input thì có cách nào không?

Lúc này bạn sẽ dễ dàng validate lần đầu tiên ở trong ngOnInit được. Nhưng như thế ở các lần sau ngOnInit không chạy lại thì cũng không phải là giải pháp toàn diện. Đây chính là lúc mà bạn có thể sử dụng life-cycle tiếp theo là ngOnChanges.

ngOnChanges sẽ chạy lại mỗi khi có một input nào bị thay đổi, nó sẽ được tự động gọi bởi Angular, do đó chúng ta có thể validate property progress như sau:

export class ProgressBarComponent implements OnInit, OnChanges {

@Input() backgroundColor: string;

@Input() progressColor: string;

@Input() progress = 0;

constructor() { }

ngOnChanges(changes: SimpleChanges) {

if ('progress' in changes) {

if (typeof changes['progress'].currentValue !== 'number') {

const progress = Number(changes['progress'].currentValue);

if (Number.isNaN(progress)) {

this.progress = 0;

} else {

this.progress = progress;

}

}

}

}

ngOnInit() {

}

}

Trong trường hợp bạn không thích dùng ngOnChanges, chúng ta có thể sử dụng getter/setter để làm điều này.

export class ProgressBarComponent implements OnInit {

@Input() backgroundColor: string;

@Input() progressColor: string;

private $progress = 0;

@Input()

get progress(): number {

return this.$progress;

}

set progress(value: number) {

if (typeof value !== "number") {

const progress = Number(value);

if (Number.isNaN(progress)) {

this.$progress = 0;

} else {

this.$progress = progress;

}

} else {

this.$progress = value;

}

}

constructor() {}

ngOnInit() {}

}

## **- @Output()**

COMPONENT INTERACTION - PARENT LISTENS FOR CHILD EVENT

Thông thường, trong một trang HTML khi có một sự kiện nào đó phát sinh ở một thẻ HTML (ví dụ sự kiện click của thẻ button, submit của form, etc) thì chúng ta sẽ có thể listen ở đâu đó trong code JavaScript.

Vậy với những Component mà chúng ta tự định nghĩa thì có cách nào bắn ra các event mà chungs ta mong muốn hay không (component event). Câu trả lời cho vấn đề này chính là EventEmitter và @Output decorator.

KHỞI TẠO COMPONENTS

Đầu tiên chúng ta sẽ cần khởi tạo một số component để minh họa như: Author List Component, Author Detail Component:

Các bạn chạy lệnh sau để tạo:

ng g c author-list

ng g c author-detail

Author List component sẽ chứa một danh sách các authors, và nó sẽ truyền vào từng author cho Author Detail hiển thị. Author Detail sẽ cho phép truyền vào input là thông tin của một author:

export interface Author {

id: number;

firstName: string;

lastName: string;

email: string;

gender: string;

ipAddress: string;

}

Author List Component

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

import { authors } from '../authors';

@Component({

selector: 'app-author-list',

template: `

<app-author-detail \*ngFor="let author of authors" [author]="author"></app-author-detail>

`,

styles: [``]

})

export class AuthorListComponent implements OnInit {

authors = authors;

constructor() { }

ngOnInit() {

}

}

Author Detail Component

import { Component, OnInit, Input } from '@angular/core';

import { Author } from '../authors';

@Component({

selector: 'app-author-detail',

template: `

<div \*ngIf="author">

<strong>{{author.firstName}} {{author.lastName}}</strong>

<button (click)="handleDelete()">x</button>

</div>

`,

styles: [``]

})

export class AuthorDetailComponent implements OnInit {

@Input() author: Author;

constructor() { }

ngOnInit() {

}

handleDelete() {}

}

Bây giờ chúng ta muốn delete author trong Author Detail component thì sao. Điều này chúng ta không nên làm, và có thể delete xong sẽ không work. Vì thực tế thông tin này không phải của Author Detail component, nên nó không được phép edit/modify/remove, mà chúng ta sẽ gửi một event cho parent component để báo rằng chúng ta muốn xóa phần tử này đó.

Lúc này bạn sẽ cần dùng đến EvenEmitter và @Output decorator

export class AuthorDetailComponent implements OnInit {

@Input() author: Author;

@Output() deleteAuthor = new EventEmitter<Author>();

constructor() { }

ngOnInit() {

}

handleDelete() {

this.deleteAuthor.emit(this.author);

}

}

Đó là cách để chúng ta gửi đi một custom event cho component. Giờ đây ở parent component có thể listen vào event trên và tương tác được với nó.

@Component({

selector: 'app-author-list',

template: `

<app-author-detail

\*ngFor="let author of authors"

[author]="author"

(deleteAuthor)="handleDelete($event)">

</app-author-detail>

`,

styles: [``]

})

export class AuthorListComponent implements OnInit {

authors = authors;

constructor() { }

ngOnInit() {

}

handleDelete(author: Author) {

this.authors = this.authors.filter(item => item.id !== author.id);

}

}

## **- Custom Two-way Binding**

## Giới thiệu lại ngModel

Như chúng ta đã biết, ngModel được cung cấp là một directive ở trong NgModule có tên là FormsModule, vậy nên để sử dụng được nó chúng ta sẽ cần imports FormsModule vào NgModule quản lý component hiện tại. Hay nói cách khác, nếu component của bạn muốn sử dụng NgModel thì bạn phải imports FormsModule và NgModule quản lý component đó.

Ví dụ, chúng ta có AppComponent và nó được quản lý bởi AppModule, vậy nên chúng ta sẽ imports FormsModule vào AppModule như sau.

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

import { BrowserModule } from '@angular/platform-browser';

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

import { AppComponent } from './app.component';

@NgModule({

imports: [ BrowserModule, FormsModule ],

declarations: [ AppComponent],

bootstrap: [ AppComponent ]

})

export class AppModule { }

Như thế là chúng ta có thể sử dụng được ngModel ở template, và sử dụng cú pháp vuông vuông tròn tròn [(ngModel)] để có được Two-way binding.

**app.component.html**

<p>

Your name: {{ name }}

</p>

<input type="text" [(ngModel)]="name">

**app.component.ts**

@Component({

selector: 'my-app',

templateUrl: './app.component.html',

styleUrls: [ './app.component.css' ]

})

export class AppComponent {

name = Nguyen Dung';

}

Giờ đây mỗi lần bạn thay đổi giá trị của input thì thẻ p sẽ hiển thị tên được sync tương ứng.

Thực chất cách viết trên là viết tắt của property binding và event binding như sau:

<input type="text" [ngModel]="name" (ngModelChange)="name = $event">

Vậy nên để tạo custom Two-way binding thì bạn chỉ cần tạo @Input và @Output với @Output có suffix là change là được, ví dụ value và valueChange.

## Toggle Component support Two-way binding

Chúng ta sẽ khởi tạo một component mới để minh họa, component có tên là toggle component:

ng g c toggle

Phần code của component sẽ được implement như sau:

**toggle.component.ts**

import { Component, OnInit, Input, Output, EventEmitter } from '@angular/core';

@Component({

selector: 'app-toggle',

templateUrl: './toggle.component.html',

styleUrls: ['./toggle.component.css']

})

export class ToggleComponent implements OnInit {

@Input() checked = false;

@Output() checkedChange = new EventEmitter<boolean>();

constructor() { }

ngOnInit() {

}

toggle() {

this.checked = !this.checked;

this.checkedChange.emit(this.checked);

}

}

**toggle.component.html**

<div class="toggle-wrapper" [class.checked]="checked" tabindex="0" (click)="toggle()">

<div class="toggle"></div>

</div>

**toggle.component.css**

.toggle-wrapper {

display: flex;

align-items: center;

justify-content: center;

cursor: pointer;

transition: all .2s;

width: 100px;

height: 100px;

border-radius: 50%;

background-color: #fe4551;

box-shadow: 0 20px 20px 0 rgba(#fe4551, 0.3);

}

.toggle-wrapper:active {

width: 95px;

height: 95px;

box-shadow: 0 15px 15px 0 rgba(#fe4551, 0.5);

}

.toggle-wrapper .toggle {

height: 17px;

width: 17px;

}

.toggle {

transition: all 0.2s ease-in-out;

height: 20px;

width: 20px;

background-color: transparent;

border: 10px solid #fff;

border-radius: 50%;

cursor: pointer;

animation: red .7s linear forwards;

}

.toggle-wrapper.checked {

background-color: #48e98a;

box-shadow: 0 20px 20px 0 rgba(#48e98a, 0.3);

}

.toggle-wrapper.checked:active {

box-shadow: 0 15px 15px 0 rgba(#48e98a, 0.5);

}

.toggle-wrapper.checked .toggle {

width: 0;

background-color: #fff;

border-color: transparent;

border-radius: 30px;

animation: green .7s linear forwards !important;

}

Component trên của chúng ta không có quá khác biệt so với những component trước đây, nó đều sử dụng những concept cơ bản như @Input, @Output, class binding, style css.

Chỉ với việc sử dụng suffix change cho property checked chúng ta có thể sử dụng component này ở bất cứ đâu với cách dùng giống như ngModel:

<app-toggle [(checked)]="checked"></app-toggle>

## **- Template variable và ViewChild/ViewChildren**

Nếu bạn cần trỏ tới một phần tử (HTMLElement/component/directive) ở trong template và thao tác trực tiếp lên nó thì sao. Có cách nào để chúng ta tạo ra một variable ở trong template và sử dụng nó không? Câu hỏi trên sẽ được trả lời trong ngày thứ 10 này.

**Parent interacts with child via local variable**

Giả sử chúng ta có AppComponent có nhúng một phần template như sau:

<app-toggle></app-toggle>

Nhưng thay vì click vào ToggleComponent để thay đổi trạng thái của nó thì chúng ta sẽ dùng một button ở parent (AppComponent) như template dưới đây:

<button (click)="doSomething">Toggle</button>

<br>

<app-toggle></app-toggle>

Làm thế nào để gọi được method toggle() của ToggleComponent?

Lúc này bạn sẽ có thể sử dụng đến Template variable như một giải pháp hữu hiệu.

<button (click)="toggleComp.toggle()">Toggle</button>

<br>

<app-toggle #toggleComp></app-toggle>

Cú pháp của Template variable chính là sử dụng #varName và bạn có thể tạo nhiều variable trong template.

Như trước đây làm với ví dụ về NgIf-Else chúng ta cũng đã sử dụng Template variable để lấy về một instance của ng-template và truyền vào cho NgIfElse như sau:

<div \*ngIf="user.age >= 13; else noPG13">

Bạn có thể xem nội dung PG-13

</div>

<ng-template #noPG13>

<div>

Bạn không thể xem nội dung PG-13

</div>

</ng-template>

**Template variable sẽ là instance của class nào?**

Thông thường, một element trong template của một component sẽ có thể là một HTMLElement, hoặc một Component. Nhưng có những trường hợp sẽ có nhiều Directives cùng được đặt vào một element, vậy Template variable sẽ cho chúng ta một object thuộc kiểu nào?

Mặc định Template variable mà không có expression #varName sẽ cố gắng lấy về một object, trong đó đối với HTMLElement thì object đó chính là HTMLElement, còn với Component thì chính là instance của Component đó.

Trong một số trường hợp bạn cần lấy chính xác một instance của một directive/component nào đó thì bạn sẽ cần sử dụng cú pháp #varName="exportAsOfDirectiveOrComponent", ví dụ như khi làm việc với FormsModule và ngModel, bạn có thể nhìn thấy cú pháp sau:

<form #nameForm="ngForm">

<input

type="text" class="form-control"

required

[(ngModel)]="model.name" name="name"

#name="ngModel">

<button>Submit</button>

</form>

Ở template trên chúng ta đã tạo ra 2 Template variable là:

* nameForm: mong muốn lấy instance của directive có exportAs là ngForm
* name: mong muốn lấy instance của directive có exportAs là ngModel

Nếu như chúng ta không request chính xác type chúng ta mong muốn là gì thì các Template variable trên chỉ lấy về HTMLElement thông thường.

## Template variable sẽ là instance của class nào?

Thông thường, một element trong template của một component sẽ có thể là một HTMLElement, hoặc một Component. Nhưng có những trường hợp sẽ có nhiều Directives cùng được đặt vào một element, vậy Template variable sẽ cho chúng ta một object thuộc kiểu nào?

Mặc định Template variable mà không có expression #varName sẽ cố gắng lấy về một object, trong đó đối với HTMLElement thì object đó chính là HTMLElement, còn với Component thì chính là instance của Component đó.

Trong một số trường hợp bạn cần lấy chính xác một instance của một directive/component nào đó thì bạn sẽ cần sử dụng cú pháp #varName="exportAsOfDirectiveOrComponent", ví dụ như khi làm việc với FormsModule và ngModel, bạn có thể nhìn thấy cú pháp sau:

<form #nameForm="ngForm">

<input

type="text" class="form-control"

required

[(ngModel)]="model.name" name="name"

#name="ngModel">

<button>Submit</button>

</form>

Ở template trên chúng ta đã tạo ra 2 Template variable là:

* nameForm: mong muốn lấy instance của directive có exportAs là ngForm
* name: mong muốn lấy instance của directive có exportAs là ngModel

Nếu như chúng ta không request chính xác type chúng ta mong muốn là gì thì các Template variable trên chỉ lấy về HTMLElement thông thường.

## Parent class with ViewChild

Vậy nếu bạn muốn gọi đến Template variable ở trong class của một component thì sao? Giải pháp ở đây là gì, vì nếu chỉ ở ngoài template thì có vẻ nó không được mạnh mẽ cho lắm, vì code logic của chúng ta nên ở trong component (và service - sau này sẽ học) thay vì ở ngoài template.

Lúc này chúng ta có thể query một Template variable ở trong Component như sau:

<button (click)="toggleInside()">Toggle inside class</button>

<br>

<br>

<app-toggle #toggleComp></app-toggle>

export class AppComponent {

@ViewChild('toggleComp') toggleComp: ToggleComponent;

toggleInside() {

this.toggleComp.toggle();

}

}

Nếu bạn sử dụng ViewChild cho một HTMLElement thì chúng ta sẽ nhận được một ElementRef thay vì một HTMLElement như sử dụng ở trong template.

<div #chartContainer></div>

export class AppComponent {

@ViewChild('chartContainer') container: ElementRef<HTMLDivElement>;

}

Lưu ý: Ngoài những thiết lập mặc định ở trên cho ViewChild, chúng ta còn có thể truyền vào config cho nó với các thông số chi tiết trong link sau:

<https://angular.io/api/core/ViewChild>

Cú pháp để sử dụng sẽ như sau:

// View queries are set before the ngAfterViewInit callback is called.

ViewChild(selector: string | Function | Type<any>, opts?: {

read?: any;

static?: boolean;

})

Trong đó các selector có thể là:

* Any class with the @Component or @Directive decorator
* A template reference variable as a string (e.g. query <my-component #cmp> with @ViewChild('cmp'))
* Any provider defined in the child component tree of the current component (e.g. @ViewChild(SomeService) someService: SomeService)
* Any provider defined through a string token (e.g. @ViewChild('someToken') someTokenVal: any)
* A TemplateRef (e.g. query with @ViewChild(TemplateRef) template;)

opts.read có thể là bất cứ một token nào - có thể là một directive, một component, một service, etc. Nếu match với token nào thì sẽ trả về. Ví dụ:

<form #nameForm="ngForm">

<input

type="text" class="form-control"

required

[(ngModel)]="model.name" name="name"

#name="ngModel">

<button>Submit</button>

</form>

export class NameFormComponent implements OnInit {

model = {

name: 'Tiep Phan'

};

@ViewChild('nameForm', {

read: ElementRef,

static: true

}) form: ElementRef<HTMLFormElement>;

constructor() { }

ngOnInit() {

console.log(this.form)

}

}

Ở trong trường hợp trên, nếu chúng ta không khai báo read thì sẽ lấy về NgForm instance, nhưng do khai báo là một ElementRef nên nó sẽ query khác với variable ở ngoài template.

opts.static nếu selector không nằm trong if/else hay một structure directive nào thì chúng ta có thể gọi nó là static: true, tức là nó không thay đổi trong suốt thời gian sống của component. Lúc này Angular (v9 trở lên) sẽ chạy phần resolve query result (tiến trình) trước khi chạy Change Detection nên chúng ta có thể truy cập nó ở trong ngOnInit như ở trên, nếu static: false (giá trị mặc định) thì tiến trình trên sẽ chạy sau khi chạy Change Detection nên bạn không thể dùng nó ở ngOnInit mà phải chạy ở ngAfterViewInit.

## Parent class with ViewChildren

Khi bạn muốn query một danh sách các element thì bạn có thể dùng ViewChildren.

ViewChildren sẽ trả về một QueryList trước khi ngAfterViewInit được chạy. Nó sẽ chứa một số property/method để chúng ta có thể listen vào một số event (Observable).

<app-toggle></app-toggle>

<br>

<app-toggle></app-toggle>

@ViewChildren(ToggleComponent) toggleList: QueryList<ToggleComponent>;

ngAfterViewInit() {

console.log(this.toggleList);

}

## **- Điều hướng nội dung: cấu hình Router**

+, Danh sách routes là các đối tượng Router, là 1 mảng các phần tử.

Định nghĩa các routes:

Const routes: Routes = [

{ path: ‘home’, component: HomeComponent },

{ path: ‘shop’, component: ShopComponent }

]

Path chính là đường dẫn trên url, client truy cập theo đường dẫn nào sẽ hiển thị trang của đường dẫn đó.

HomeComponent: đưa ra component mong muốn đến đường dẫn path tương ứng

Ví dụ: nếu path là ‘home’, sẽ load component là HomeComponent

+, imports: [RouterModule.forRoot(routes)]: Công khai module đó ra bên ngoài để cho client có thể gọi đến

+, Để client có thể gọi đến url ta cần cấu hình trên thẻ html. Sử dụng thuộc tính routerLink=”/home”

+ Cuối cùng cần nơi để load các component theo phần router vừa cấu hình bằng cách sử dụng <router-outlet></<router-outlet>, được coi như 1 mặt phẳng để chứa các component lên cho client xem

{path: '', redirectTo: 'home', pathMatch: 'full'},

Mặc định khi đường dẫn url là trống (tương ứng khi mới truy cập vào website) sẽ để redirectTo trang home

+, Khi client cố ý gõ 1 đường dẫn không hề có, cần điều hướng đến 1 trang Not Found và cấu hình trong router như sau:

{path: '\*\*', component: NotFoundComponent}

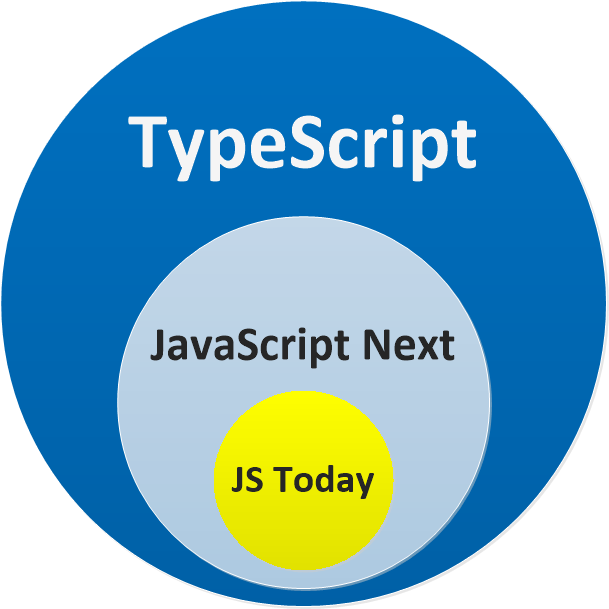
+, Để cấu hình đường dẫn con của 1 đường dẫn cần them thuộc tính children, trong children chứa 1 tập các đường dẫn khác như đường dẫn cha.

Ví dụ:

## **- Typescript Data Type**

**TypeScript là gì?**

**TS** là một **superset** của **JavaScript** (**JS**). Khi các bạn cài đặt **TS** vào hệ thống của mình, các bạn sẽ có quyền truy xuất đến được 1 **Command Line Interface** (**CLI**) gọi là **TypeScript Compiler** (tsc). Nhiệm vụ của tsc chính là *compile* code **TS** về **JS** để trình duyệt (hoặc runtime environment như **NodeJS**) có thể hiểu và xử lý được.

Để có cái nhìn bao quát hơn về **superset**, các bạn xem hình sau: [](https://github.com/angular-vietnam/100-days-of-angular/blob/master/assets/typescript-graphics.png)

Theo như hình minh hoạ thì **TS** = **JS** plus (+) **something else**.

**TypeScript minus (-) "something else"**

Đầu tiên và quan trọng nhất là **TS** cung cấp **Static Types**. **JS** là một ngôn ngữ mang tính chất Dynamic và vì thế nên **JS** rất *thoải mái*, nếu không muốn nói là *quá thoải mái*. Các bạn xem qua ví dụ sau:

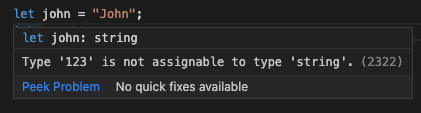
let john = 'John';

john = 123;

Các bạn có thấy gì lạ không? Đó là biến john được khởi tạo là 1 chuỗi "John" và sau đó lại được gán cho 1 số 123. Điều này hoàn toàn hợp lệ trong **JS**. Nhìn qua thì có vẻ *thoải mái*, nhưng rất dễ dẫn đến những lỗi ngớ ngẩn khi các bạn phát triển phần mềm, nếu như không cẩn thận.

Cùng 1 đoạn code trên nhưng trong **TS** thì **TS** trước tiên sẽ *tự cung cấp type* cho biến john (khi được gán giá trị "John") một primitive type là string vì giá trị "John" là dạng chuỗi (string). Khi giá trị 123 được gán cho biến john thì **TS** sẽ bắt lỗi tại thời điểm code được *compile* (**Compilation Time Error**) vì 123 (number) thì không thể gán cho 1 biến có type là string được.

Từ đoạn này trở xuống, mình sẽ gọi việc *tự cung cấp type* là **infer**

[](https://github.com/angular-vietnam/100-days-of-angular/blob/master/assets/ts-compilation-error.png)

Lỗi trên là **Compilation Time Error**. Các bạn sẽ thường gặp lỗi này ở thời điểm *viết code*. Tuy nhiên, việc các bạn nhận được error ở thời điểm *viết code* thì đó là nhờ TextEditor/IDE trợ giúp chứ trên thực tế, đoạn error trên vẫn là 1 **Compilation Time Error** nhé.

**Default Types**

Sau đây là danh sách các **default types** trong **TS**

let someString: string;

let someNumber: number;

let someBoolean: boolean;

let something: any; // có thể gán sang cho bất kỳ kiểu dữ liệu nào khác

let someStringArray: string[]; // tương tự cho number[], boolean[], any[]

let someObject: object;

let someNull: null;

let someUndefined: undefined;

let someUnknown: unknown;

let someNever: never; // ví dụ như một hàm throw exception

let someTuple: [string, number];

let someVoidFunction: () => void; // một hàm không trả về giá trị gì sau khi thực thi

let someFunction: () => string; // một hàm trả về giá trị có type "string" sau khi thực thi

someVoidFunction và someFunction đều có type là Function nhưng mình ghi cả 2 để cho các bạn thấy được type void.

**Interface/Type**

**TS** cung cấp Interface và Type để các bạn có thể định nghĩa được **type** cho một đối tượng (object)

interface User {

firstName: string;

lastName: string;

age: number;

job?: string;

}

// hoặc dùng type. Chỉ nên dùng 1 trong 2 cho cùng 1 tên (ở đây là User)

type User = {

firstName: string;

lastName: string;

age: number;

job?: string;

};

const john: User = {

firstName: 'John',

lastName: 'Doe',

age: 20,

job: 'Student',

};

const susan: User = {

firstName: 'Sue',

lastName: 'Smith',

age: 40,

};

Ở đoạn code trên, các bạn sẽ thấy chúng ta khai báo một interface User bao gồm firstName, lastName, age, và job.

Các bạn hiểu nôm na interface giống như một cái *khuôn bánh* vậy, các bạn muốn *bánh* của mình có hình dạng như thế nào thì chúng ta sẽ sử dụng cái *khuôn* có hình dạng như vậy.

Kế tiếp, các bạn để ý cú pháp job?: string nhé. Đây gọi là Optional Property. Ở trong User, job là một property **không-bắt-buộc**, nghĩa là có cũng được, không có cũng không sao trong khi 3 properties còn lại là **bắt buộc phải có**.

Việc sử dụng interface sẽ giúp cho trình soạn thảo (text editor) gợi ý cho object của các bạn có những properties gì trên object đó. Khi type john., text edior sẽ gợi ý được : firstName, lastName, age, và job cho các bạn lựa chọn khi truy xuất. Điều này giảm thiểu việc sai lỗi chính tả khi các bạn viết code.

Interface và Type trong nhiều trường hợp có thể sử dụng qua lại được. Tuy nhiên, các bạn nên chọn 1 trong 2 và giữ chuẩn này trong toàn bộ dự án của mình. Interface và Type có 1 số điểm khác biệt, các bạn có thể xem thêm tại: [interface-vs-type](https://medium.com/@martin_hotell/interface-vs-type-alias-in-typescript-2-7-2a8f1777af4c)

**Class**

class là cú pháp xuất hiện trong **ES2015** hay **ES6** (và xuất hiện rất nhiều ở những ngôn ngữ lập trình hướng đối tượng khác). Trong **JS**, class thực chất chỉ là *syntactic sugar* cho **Prototypal Inheritance** ([Prototypal Programming](https://en.wikipedia.org/wiki/Prototype-based_programming), hay các bạn hay gọi trong **JS** là **prototype chain**).

**TS** đem lại support hàng đầu cho class với hệ thống typings khá mạnh mẽ. Để hiểu rõ hơn về class, các bạn cần có kiến thức về **Object Oriented Programming** (**[OOP](https://en.wikipedia.org/wiki/Object-oriented_programming)**). Ở đây, chúng ta sẽ chỉ khám phá qua các cú pháp dùng class trong **TS**

class User {

firstName: string;

lastName: string;

age: number;

job?: string;

constructor(firstName: string, lastName: string, age: number, job?: string) {

this.firstName = firstName;

this.lastName = lastName;

this.age = age;

this.job = job;

}

}

**TS** cũng hỗ trợ **Access Modifier** cho class với các keywords như: public, private, và protected. Kết hợp với các **Access Modifer**, các bạn có thể khai báo class User với cú pháp ngắn gọn sau:

class User {

constructor(

public firstName: string,

public lastName: string,

public age: number,

public job?: string

) {}

}

Tới đây, chắc chắn sẽ có thắc mắc là dùng class hay dùng interface. Các bạn đọc qua bài viết này nhé: [class-vs-interface-typescript](https://ultimatecourses.com/blog/classes-vs-interfaces-in-typescript)

**Generics**

Generics là một trong những tính năng cực kỳ tốt mà **TS** mang lại. Generics sẽ giúp các bạn viết code theo hướng **Abstraction** dễ dàng hơn, theo sát hơn **SOLID Principles** (**[SOLID](https://en.wikipedia.org/wiki/SOLID)**)

abstract class BaseService<T> {

protected model: Model<T>;

find(): T[] {

return this.model.findAll();

}

findOne(id: number): T {

return this.model.findById(id);

}

}

class DogService extends BaseService<Dog> {

constructor(dogModel: Model<Dog>) {

super();

this.model = dogModel;

}

}

class CatService extends BaseService<Cat> {

constructor(catModel: Model<Cat>) {

super();

this.model = catModel;

}

}

Ví dụ trên là một trong những cách áp dụng **Abstraction** vào code của các bạn.

Ở đây, mình viết 2 hàm find() và findOne() trong abstract class BaseService.

Cú pháp <T> chính là Generics, hay còn gọi là Type Parameter.

BaseService nhận vào 1 Type Parameter gọi là T.

Cũng như parameter thông thường, các bạn có thể đặt bất cứ tên gì: <T>, <K>, <Type>, <Props> ..v.v..

Khi mình cho DogService và CatService extends BaseService, thì trên DogService sẽ có 2 hàm: find() và findOne() với đúng type Dog mà mình ko cần phải viết lại 2 hàm trên trong DogService. (tương tự, CatService cũng vậy, và với đúng type là Cat).

Trên đây chỉ là một ví dụ nhỏ. Để hiểu và áp dụng nhiều, các bạn phải luyện tập và hiểu **Object-Oriented Programming**

**Tại sao lại là TypeScript?**

**Pros**

Như những gì đã nhắc đến ở trên, **TS** giúp developers phát triển phần mềm một cách tường minh hơn với những type definition mà **TS** cũng như các thư viện dùng cho **TS** cung cấp. Với khả năng áp dụng tính thừa kế với những syntax quen thuộc trong **OOP** như: abstract, class, và type parameter <T>, **TS** giúp developers có thể phát triển ứng dụng một cách nhanh, tường minh, chính xác, dễ bảo trì và mở rộng hơn.

Trong thời gian trở lại đây, **TS** luôn có thứ hạng rất cao (hạng 1 hoặc hạng 2) về **Ngôn ngữ được yêu thích** trong những cuộc khảo sát có tiếng tăm như **[StackOverflow Survey](https://insights.stackoverflow.com/survey/2020" \l "technology-most-loved-dreaded-and-wanted-languages-loved)**.

**Cons**

Nhưng cũng như bất cứ thứ gì trên đời (bất cứ không nhỉ? 🤨) thì đều có mặt tốt mặt xấu.

**TS** mang lại nhiều lợi ích, nhưng cũng mang lại không ít phiền toái. Điển hình như:

* code nhiều hơn vì những định nghĩa types (type definition)
* conditional check nghiêm khắc hơn gây khó khăn và các thư viện bên ngoài phải có type defs (d.ts file) để hỗ trợ **TS**, nếu không thì cũng như không….
* [TypeScript Tax](https://medium.com/javascript-scene/the-typescript-tax-132ff4cb175b).

**Trade-off**

Nhưng lợi ích của **TS** mang lại thực sự là lớn hơn so với phiền phức. Việc **Angular** chọn **TS** làm ngôn ngữ chính cũng giúp chúng ta thấy được tầm quan trọng về **structure**, **maintainability**, và **scalability** của chính **Angular**. Chúng ta muốn có gì đó thì phải đánh đổi một thứ khác.

## **- Typescript Advanced**

**Union Type**

Union Type là những types mang tính chất: **EITHER OR** (tạm dịch là **Hoặc cái này Hoặc cái kia**). Để viết Union Type, chúng ta dùng **Pipe Symbol** (|).

Ví dụ, chúng ta có 1 hàm listen() như sau:

function listen(port: unknown) {

if (typeof port === 'string') {

port = parseInt(port, 10);

}

server.listen(port);

}

**typeof**

Ở ví dụ trên, chúng ta gặp 1 cú pháp lạ typeof. typeof là 1 operator dùng để lấy về type của 1 biến. Giá trị mà typeof trả về luôn có type là string

typeof 'string'; // string

typeof 123; // number

typeof true; // boolean

typeof {}; // object

typeof []; // object

typeof (() => {}); // function

typeof null; // object

typeof undefined; // undefined

typeof new Date(); // object

typeof String; // function

typeof Boolean; // function

typeof NaN; // number

typeof typeof 123; // string

Quay trở lại hàm listen(), hàm listen() nhận vào 1 tham số port có type là unknown, nghĩa là chúng ta truyền vào 1 tham số với type nào đó mà chúng ta *chưa biết* (unknown) tại thời điểm viết code. Nói cách khác, hàm listen() trên có thể nhận: string, number, boolean, array, object, và function (kể cả undefined và null).

unknown là type được giới thiệu trong **TS 3.0**. **TS** khuyến cáo sử dụng unknown thay vì any ở nhiều trường hợp các bạn muốn có 1 type *chưa biết type lúc code* nhưng hạn chế *type chưa biết* này. Các bạn nên đọc thêm về unknown: [new unknown top type](https://www.typescriptlang.org/docs/handbook/release-notes/typescript-3-0.html" \l "new-unknown-top-type)

Vậy là chúng ta đã biết unknown là type như thế nào: bất cứ giá trị nào cũng có thể gán được cho unknown. Điều này thật sự không hay tí nào, vì implementation của hàm listen() chỉ xử lý 2 types là string và number mà thôi. Giờ xử lý như thế nào? Cách tốt nhất là dùng Union Type. Chúng ta sẽ viết lại hàm listen() như sau:

function listen(port: string | number) {

// do listen

}

listen('3000'); // ok

listen(3000); // ok

listen(true); // TypeError: Argument of type true is not assignable to parameter type string | number

listen(); // TypeError: Invalid number of arguments, expected 1

**TS** *la làng* (Compilation Time Error) ngay nếu như truyền vào 1 boolean (không phải string HOẶC number) hoặc không truyền vào gì khi gọi hàm listen().

Khi chạy tsc để compile code **TS** sang **JS**, các bạn cũng sẽ gặp Compilation Error tương tự. Tuy nhiên, code **JS** cũng sẽ được compiled ra đàng hoàng, các bạn cũng sẽ chạy được code **compiled JS** kia nhưng sẽ gặp lỗi ở runtime hay còn gọi là Runtime Error. Phân biệt 2 loại errors này nhé 🙂

Tương tự như tạo Union Type cho tham số, các bạn cũng có thể tạo Union Type cho giá trị trả về của hàm nhé.

function getSomething(): string | number {

return 'string'; // works

return 30; // works

return true; // TypeError: Returned expression type boolean is not assignable to type string | number

}

Để tái sử dụng (reuse) 1 Union Type bất kỳ, các bạn có thể tạo Type Alias cho Union Type đó

type StringOrNumber = string | number;

function listen(port: StringOrNumber) {...}

function getSomething(): StringOrNumber {...}

**Intersection Type**

Ngược với Union Type, Intersection Type là type mà kết hợp nhiều type lại với nhau. Nói cách khác, Intersection Type là type có tính chất: **AND** (dịch nôm na là **và**).

function merge<T1, T2>(o1: T1, o2: T2): T1 & T2 {

return { ...o1, ...o2 };

}

merge({ foo: 'bar' }, { bar: 'foo' });

Hàm merge({ foo: 'bar' }, { bar: 'foo' }) này sẽ có giá trị trả về là { foo: string } & { bar: string }.

Ngoài những cách sử dụng thông thường trong function hoặc trong những đoạn code mà khái niệm **OOP** thông thường không áp dụng được thì Intersection Type được dùng rất nhiều trong việc thiết kế hệ thống type cho những thư viện UI Components. Ví dụ:

type StyleProp = {

backgroundColor: string;

color: string;

margin: string;

padding: string;

...

}

type ButtonProps = {

onClick: (event: MouseEvent) => void;

} & StyleProps;

type TextProps = {

fontSize: string;

fontWeight: number;

...

} & StyleProps;

Những Component này có những type Style khác nhau, nhưng cũng có những type cơ bản giống nhau. Ví dụ như Text sẽ có thêm fontSize, fontWeight còn Button sẽ có onClick. Tác giả của những thư viện UI này sẽ sử dụng Intersection Type để viết thư viện UI của họ mà không phải lặp đi lặp lại nhiều 1 số type giống nhau. Cách dùng Intersection Type này còn có tên gọi khác là Type Composition.

Type Composition là 1 chủ đề rất hay, và rộng lớn trong **TS**. Các bạn nên google để tự tìm hiểu thêm nhé.

**Conditional Type**

Conditional Type có mặt trong **TS** từ version 2.8 và có thể nói đây là một trong những tính năng nổi bật nhất của **TS**. Conditional Type, đúng như tên gọi của nó, giúp cho chúng ta có thể tạo ra những type theo điều kiện. Điều này dẫn đến 1 hệ thống type cực kỳ linh hoạt (*robust*) mà **TS** mang lại cho người dùng. Ví dụ:

T extends U ? X : Y;

Đoạn code trên có thể hiểu nôm na là khi type T có thể gán được cho type U thì sẽ trả về type X, còn không thì trả về type Y.

**Type Alias**

Type Alias có thể hiểu là *alias* (tên thay thế) một hoặc nhiều loại types nào đó thành 1 type, giống như StringOrNumber phía trên. StringOrNumber là 1 Type Alias của string | number (Union Type). Type Alias có thể dùng cho bất cứ loại type nào.

**Type Alias và Union Type**

Sau đây, chúng ta sẽ cùng xem qua thêm 1 ví dụ về Type Alias dùng cho Union Type nhé. Tưởng tượng chúng ta cần tạo 1 component Flex và component này có những yêu cầu cơ bản sau:

* Flex sẽ có style mặc định là `display: 'flex'
* Flex sẽ nhận vào 1 Input flexDirection để có thể gán vào style như sau: flex-direction: flexDirection

@Component({

selector: 'flex-container',

template: `<ng-content></ng-content>`,

})

export class FlexComponent {

@Input() flexDirection: string = 'row';

@HostBinding('style.display') get display() {

return 'flex';

}

@HostBinding('style.flex-direction') get direction() {

return this.flexDirection;

}

}

HostBinding là khái niệm chúng ta chưa tìm hiểu trong chuỗi 100 ngày này. Các bạn chỉ cần hiểu là chúng ta dùng HostBinding để bind giá trị lên selector tag <flex-container></flex-container>

Như đa số các bạn đã biết, flexDirection của một **flex container** sẽ có thể có 1 trong 4 gía trị: column, row, column-reverse, và row-reverse. Nhưng ở đoạn code trên, chúng ta có thể truyền vào bất cứ 1 giá trị string nào cho flexDirection. Và điều này sẽ dẫn đến style của component này sẽ không được chính xác. Một lần nữa để ngăn ngừa việc truyền tham số tràn lan, chúng ta dùng Union Type

type FlexDirection = 'row' | 'column' | 'row-reverse' | 'column-reverse';

@Component({

selector: 'flex-container',

template: `<ng-content></ng-content>`

})

export class FlexComponent {

@Input() flexDirection: FlexDirection = 'row';

@HostBinding('style.display') get display() {...}

@HostBinding('style.flex-direction') get direction() {

return this.flexDirection;

}

}

Và cách dùng:

<!-- app.component.html -->

<flex-container>

<button>Submit</button>

<button>Cancel</button>

</flex-container>

<flex-container flexDirection="column">

<input type="email" />

<input type="password" />

</flex-container>

Khi dùng flex-container component trên template, **TS** đã có thể gợi ý (intellisense) 4 giá trị của flexDirection khi các bạn muốn truyền gia trị cho flexDirection

**Type Alias và Conditional Type**

Kế tiếp, chúng ta sẽ tìm hiểu 1 ví dụ về Type Alias và Conditional Type nhé.

type ObjectDictionary<T> = { [key: string]: T };

type ArrayDictionary<T> = { [key: string]: T[] };

export type Dictionary<T> = T extends []

? ArrayDictionary<T[number]>

: ObjectDictionary<T>;

type StringDictionary = Dictionary<string>; // {[key: string]: string}

type NumberArrayDictionary = Dictionary<number[]>; // {[key: string]: number[]}

type UserEntity = Dictionary<User>; // {[key: string]: User}

Ở ví dụ trên, chúng ta có 3 Type Alias: ObjectDictionary, ArrayDictionary, và Dictionary. Trong đó, Dictionary có thể được xem là **True Type** (type được export ra cho bên ngoài sử dụng), còn ObjectDictionary và ArrayDictionary có thể được xem là **Support Type** (type dùng để hỗ trợ cho **True Type**). Và code thì khá dễ hiểu, nếu mình truyền vào 1 type dạng number[] cho type parameter T ở Dictionary<T> thì T extends [] sẽ được đính giá (evaluate) là truthy và Dictionary<number[]> sẽ trả về type ArrayDictionary<number>

Với Type Alias và Conditional Type, **TS** ngoài việc cung cấp cho chúng ta khả năng tạo những dạng type thú vị như trên và kết hợp chúng lại với nhau, thì **TS** còn cung cấp cho chúng ta 1 số *built-in* type rất hay. Chúng ta cùng điểm qua một số *built-in* types hay dùng nhé:

* Exclude
* Extract
* Readonly
* Partial
* Nullable
* Pick
* Record
* ReturnType
* Omit (Omit là Type Alias của Pick và Exclude)

// Exclude/Extract

type Exclude<T, U> = T extends U ? never : T;

type Extract<T, U> = T extends U ? T : never;

// Exclude: Loại bỏ những types ở T nếu như những types này gán được cho U

type SomeDiff = Exclude<'a' | 'b' | 'c', 'c' | 'd'>; // 'a' | 'b'. 'c' đã bị removed.

// Extract: Loại bỏ những types ở T nếu như những types này KHÔNG gán được cho U

type SomeFilter = Extract<'a' | 'b' | 'c', 'c' | 'd'>; // 'c'. 'a' và 'b' đã bị removed.

// hoặc có thể dùng Exclude để tạo type alias NonNullable

type NonNullable<T> = Exclude<T, null | undefined>; // loại bỏ null và undefined từ T

type Readonly<T> = { readonly [P in keyof T]: T[P] }; // làm tất cả các props trong type thành readonly. Eg: Rất có lợi khi dùng cho Redux State.

type Partial<T> = { [P in keyof T]?: T[P] }; // làm tất cả các props trong type thành optional. Eg: Rất có lợi cho việc type 1 tham số khi cần truyền vào 1 type nào đó mà ko bắt buộc.

type Nullable<T> = { [P in keyof T]: T[P] | null }; // cái này tương tự như Partial, Partial sẽ trả về T[P] | undefined. Còn Nullable sẽ trả về T[P] | null

type Pick<T, K extends keyof T> = { [P in K]: T[P] };

type Record<K extends keyof any, T> = { [P in K]: T };

// Pick: Pick từ trong T những type có key là K

type Person = {

firstName: string;

lastName: string;

password: string;

};

type PersonWithNames = Pick<Person, 'firstName' | 'lastName'>; // {firstName: string, lastName: string}

// Record: Gán type T cho lần lượt từng key P trong K

type ThreeStringProps = Record<'prop1' | 'prop2' | 'prop3', string>;

// { prop1: string, prop2: string, prop3: string }

type ReturnType<T> = T extends (...args: any[]) => infer R ? R : any;

// ReturnType: trả về type của giá trị mà function T trả về.

type Result = ReturnType<() => string>; // string

type Omit<T, K extends keyof T> = Pick<T, Exclude<keyof T, K>>;

// Omit: loại bỏ type có key là K trong T

type PersonWithoutPassword = Omit<Person, 'password'>; // {firstName: string, lastName: string}

**- Content Projection trong Angular**

Có những khi trong quá trình phát triển ứng dụng với Angular chúng ta có thể sẽ gặp một số dạng component giống nhau về phần layout chẳng hạn, điểm khác biệt chỉ là một số label, content bên trong nó. Lúc này rất dễ để các bạn có thể tạo một component có nhận vào các Input, và render dựa vào các Input đó.

Nhưng có bao giờ bạn tự hỏi, những gì chúng ta đặt vào giữa cặp mở/đóng tag sẽ đi về đâu chưa? Liệu có cách nào để tạo component mà việc cần làm chỉ là thêm một số template vào giữa cặp tag đó thì sẽ có những thứ chúng ta mong muốn không.

Câu trả lời cho bài toán này có thể dùng đến Content Projection trong Angular.

## ng-content và những câu hỏi

### Làm thế nào để sử dụng ng-content?

Giả sử trong ứng dụng của chúng ta sẽ sử dụng lại component Toggle từ những buổi học trước để làm phần khảo sát khách hàng. Các câu hỏi sẽ ở dạng Yes/No, và nội dung label sẽ khác nhau cho từng câu hỏi. Vậy làm thế nào để chúng ta biến component Toggle trở nên linh động hơn mà không cần thêm Input nào. Có thể sử dụng ngay content được truyền vào (project) ở giữa cặp thẻ mở/đóng được không?

Đó là một perfect use-case cho ng-content có thể được sử dụng. Bạn chỉ cần đặt ng-content vào bất kỳ đâu trong template của component là được.

**toggle.component.html**

<div

class="toggle-wrapper"

[class.checked]="checked"

tabindex="0"

(click)="toggle()"

>

<div class="toggle"></div>

</div>

<div class="toogle-label">

<ng-content></ng-content>

</div>

**app.component.html**

<app-toggle [(checked)]="questions.question1">

<span>Question 1</span>

</app-toggle>

<app-toggle [(checked)]="questions.question2">

<span>Question 2</span>

</app-toggle>

### Sử dụng multiple ng-content được không?

Bạn có thể nghĩ rằng nếu mình đặt ng-content hai lần ở trên template thì sẽ thế nào? liệu nó có hiển thị gấp đôi số phần tử hay không?

**toggle.component.html**

<div class="toogle-label">

<div>content 1</div>

<ng-content></ng-content>

</div>

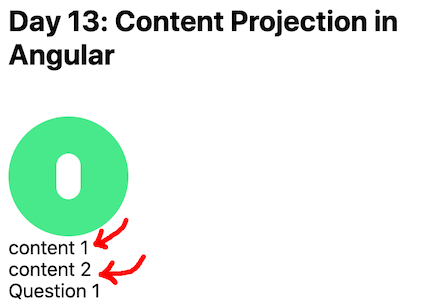
<div class="toogle-label">

<div>content 2</div>

<ng-content></ng-content>

</div>

Khi nhìn vào kết quả render, chúng ta sẽ thấy chỉ có content 2 là được hiển thị với label mà chúng ta truyền vào. Như vậy nếu sử dụng nhiều lần ng-content sẽ dẫn đến kết quả có thể không như chúng ta mong muốn. Điều này là hoàn toàn bình thường, giống như bao thẻ html khác như header, chúng ta chỉ có duy nhất 1 slot để hiển thị. Vậy nên đối với ng-content ở dạng trên, chúng ta chỉ nên có một tag duy nhất.

[](https://github.com/angular-vietnam/100-days-of-angular/blob/master/assets/ng100doc-d013-multiple-ng-content.png)

### ng-content và selector

Khi để ý thẻ table của html các bạn sẽ thấy rằng, dù thead, tbody, tfoot có đặt ở bất kỳ thứ tự nào trong thẻ table, nó đều được đưa về đúng thứ tự là thead rồi đến tbody, sau cùng là tfoot. Điều này các bạn cũng có thể làm tương tự với ng-content kết hợp **selector**. Ngoài việc **project** có thứ tự ra, nó còn cho phép chúng ta dùng nhiều ng-content.

What? Bên trên chúng ta mới bảo không nên dùng nhiều ng-content cơ mà.

Đúng vậy, nhưng không phải là ng-content ở dạng select all như trên.

Các dạng của selector có thể bao gồm:

* Tag selector: <ng-content select="some-component-selector-or-html-tag"></ng-content>
* CSS Class selector: <ng-content select=".some-class"></ng-content>
* Attribute selector: <ng-content select="[some-attr]"></ng-content>
* Combine nhiều selectors: <ng-content select="some-component-selector-or-html-tag[some-attr]"></ng-content>

Ví dụ với Toggle component:

**toggle.component.html**

<header>

<ng-content select=".toogle-header"></ng-content>

</header>

<div

class="toggle-wrapper"

[class.checked]="checked"

tabindex="0"

(click)="toggle()"

>

<div class="toggle"></div>

</div>

<div class="toogle-label">

<ng-content select="label"></ng-content>

</div>

<div class="toggle-content">

<ng-content></ng-content>

</div>

**app.component.html**

<app-toggle [(checked)]="questions.question1">

<h3 class="toogle-header">Header 1</h3>

<label>Question 1</label>

<span>Some paragraph</span>

</app-toggle>

Và dù ở **app.component.html** chúng ta có đặt sai thứ tự thì Toggle component vẫn sẽ hiển thị như những gì chúng ta mong muốn.

<app-toggle [(checked)]="questions.question2">

<h3 class="toogle-header">Header 2</h3>

<span>Some paragraph 2</span>

<label>Question 2</label>

</app-toggle>

Lưu ý: khi sử dụng selector nếu chúng ta project vào nhiều elements thỏa mãn selector đó thì ng-content select sẽ nhận hết tất cả các elements.

### ng-content và ngProjectAs

Giả sử Toggle component mong muốn nhận vào một component có selector là app-label, và chúng ta sẽ cung cấp một component app-label với nhiều tính năng rất xịn xò.

<div class="toogle-label">

<ng-content select="app-label"></ng-content>

</div>

Nhưng người khác khi sử dụng Toggle component lại muốn sử dụng một component label khác, hay đơn giản chỉ là dùng tag label của HTML, hoặc component app-label không phải là con trực tiếp của app-toggle mà nó bị wrap bởi một thẻ div chẳng hạn.

Lúc này, selector kia của chúng ta sẽ không thể nhận dạng được. Vậy có cách nào để báo cho Angular biết rằng chúng ta đang mong muốn project content này hay không?

Cứu cánh chính là ngProjectAs, nó là một cách khai báo tường minh để Angular biết lối mà xử lý.

**app.component.html**

<app-toggle [(checked)]="questions.question1">

<h3 class="toogle-header">Header 1</h3>

<label ngProjectAs="app-label">Question 1</label>

<span>Some paragraph</span>

</app-toggle>

**- ng-template và ng-container**

## ng-template

Trong [bài viết thứ tư](https://github.com/angular-vietnam/100-days-of-angular/blob/master/Day004-Structure-Directive-If-Else.md) của series, anh @tiepphan đã nói có nói đến một trường hợp dùng ng-template. Đó là khi dùng \*ngIf với điều kiện else, chúng ta có thể truyền vào một template reference đc định nghĩa thông qua cú pháp #templateReferenceName để render lên UI.

<div \*ngIf="user.age >= 13; else noPG13">

Bạn có thể xem nội dung PG-13

</div>

<ng-template #noPG13>

<div>

Bạn không thể xem nội dung PG-13

</div>

</ng-template>

Thông qua ví dụ trên, chắc các bạn cũng đã nhận ra được đôi điều:

* Khi code HTML của bạn dc bao quanh bởi ng-template, phần HTML đó sẽ không dc render lên UI ngay lập tức. Mà chỉ dc render trong một số trường hợp, ví dụ như khi \*ngIf else tmpl hoặc thông qua ngTemplateOutlet mà chúng ta sẽ đề cập đến ở phần sau của bài viết.
* Tên gọi của ng-template cũng phần nào nói lên đc ý nghĩa của nó. Template hiểu nôm na là mẫu, dạng. Dịch ra tiếng Việt hơi khó, tuy nhiên khi kết hợp nhiều template với nhau thì chúng ta có thể có một UI đầy đủ.

Từ những điểm trên, có thể định nghĩa ng-template là một thành phần của Angular để render HTML code. Và phần HTML code nằm trong ng-template không bao giờ được hiển thị trực tiếp ở nơi nó được định nghĩa

### Khi nào thì nên dùng ng-template?

Một số trường hợp hay cần dùng đến ng-template theo như kinh nghiệm của mình.

#### 1. Dùng kết hợp với các Structure Directive của Angular, ví dụ như \*ngIf

#### 2. Khi một số UI element trong một component bị lặp lại trong chính component đó, nhưng phần code đó quá nhỏ để tách ra làm một component riêng.

Ví dụ như bạn có một component có chứa biến một biển tên là counter. Phần UI của counter này sẽ đc lặp lại ở trong component của bạn vài lần với UI giống nhau.

Đây là cách bình thường chúng ta hay làm. Copy and paste code.

<div class="card">

<div class="card-header">

You have selected <span class="badge badge-primary">{{ counter }}</span> items.

</div>

<div class="card-body">

There are <span class="badge badge-primary">{{ counter }}</span> items was selected.

</div>

<div class="card-footer">

You have selected <span class="badge badge-primary">{{ counter }}</span> items.

</div>

</div>

Và đây là cách chúng ta có thể viết lại bằng cách dùng ng-template và ngTemplateOutlet.

<div class="card">

<div class="card-header">

You have selected <ng-container [ngTemplateOutlet]="counterTmpl"></ng-container>.

</div>

<div class="card-body">

There are <ng-container [ngTemplateOutlet]="counterTmpl"></ng-container> was selected.

</div>

<div class="card-footer">

You have selected <ng-container [ngTemplateOutlet]="counterTmpl"></ng-container>.

</div>

</div>

<ng-template #counterTmpl>

<span class="badge badge-primary">{{ counter }}</span> items

</ng-template>

Khi viết lại code dùng ng-template, ưu điểm dễ nhận thấy đó là:

* Nếu cần sửa lại UI cho counter. Thay vì phải sửa ở 3 nơi, bây giờ ta chỉ cần sửa ở một vị trí đó là ng-template của counter thôi. Tránh những lỗi typo hay find and replace bị thiếu.
* Vì phần template này chỉ gói gọn trong đúng một dòng code nên dùng ng-template tiện hơn hẳn là phải tách phần counter ra một component mới.

#### 3. Dùng ng-template để pass vào component khác. Hỗ trợ override template có sẵn trong component.

Ví dụ mình có component tab-container, mặc định sẽ render tab với template default là defaultTabButtonsTmpl.

@Component({

selector: 'tab-container',

template: `

<ng-template #defaultTabButtonsTmpl>

<div class="default-tab-buttons">

...

</div>

</ng-template>

<ng-container \*ngTemplateOutlet="headerTemplate || defaultTabButtons"></ng-container>

... rest of tab container component ...

`

})

export class TabContainerComponent {

@Input() headerTemplate: TemplateRef<any>; // Custom template provided by parent

}

Tuy nhiên, khi dùng tab-container bạn hoàn toàn có thể pass vào template mới để override lại default UI ở parent component.

@Component({

selector: 'app-root',

template: `

<ng-template #customTabButtons>

<div class="custom-class">

<button class="tab-button" (click)="login()">

{{loginText}}

</button>

<button class="tab-button" (click)="signUp()">

{{signUpText}}

</button>

</div>

</ng-template>

<tab-container [headerTemplate]="customTabButtons"></tab-container>

`

})

## ngTemplateOutlet

Qua ví dụ trên thì có thể thấy ngay ngTemplateOutlet là cách dùng để render một template được tạo ra bởi ng-template lên UI. Cú pháp như sau

* \*ngTemplateOutlet="templateRef"(chú ý dấu sao \* nhé, không có là không chạy đâu đấy)
* [ngTemplateOutlet]="templateRef"

Tuy nhiên, như component có @Input() để truyền data từ bên ngoài vào, thì ng-template cũng có cú pháp tương tự để truyền data. Đó chính là ngTemplateOutletContext

Ví dụ như mình muốn reuse lại một button với những tên khác nhau, ngoài ra, một button mình muốn có icon, và một button mình ko muốn có icon. Thì bạn hoàn toàn cũng có thể dùng kết hợp giữa ng-template, ngTemplateOutlet và ngTemplateOutletContext để viết code.

<button class="btn btn-primary">

Click here

</button>

<button class="btn btn-danger">

<i class="fa fa-remove"></i>

Delete

</button>

Hoàn toàn có thể được viết lại thành.

<ng-template #buttonTmpl

let-label="label"

let-className="className"

let-icon="icon">

<button [ngClass]="['btn', className ? className : '']">

<i \*ngIf="icon"

class="fa {{icon}}"></i>

{{ label }}

</button>

</ng-template>

<ng-container [ngTemplateOutlet]="buttonTmpl"

[ngTemplateOutletContext]="{ label: 'Click here', className: 'btn-primary', icon: null }">

</ng-container>

<ng-container [ngTemplateOutlet]="buttonTmpl"

[ngTemplateOutletContext]="{ label: 'Remove', className: 'btn-danger', icon: 'fa-remove' }">

</ng-container>

Tuy nhiều code hơn, nhưng giờ chúng ta có thể reuse lại button này trong cùng component. Hoặc pass cái template này vào component khác cũng được, ko vấn đề gì cả.

Vài điểm chú ý:

* Khi define ra ng-template, bạn có thể config cho template đó nhận vào value bằng cách dùng cú pháp let-name="name". Trong đó name ở bên trái dấu bằng là variable bạn có thể access được ở trong ng-template, còn name ở bên phải dấu bằng là tên của object property khi pass qua ngTemplateOutletContext. Hai cái name này hoàn toàn có thể khác nhau, không có vấn đề gì cả.
* Nếu bạn không đặt tên cho variable ở vế phải của dấu bằng, chỉ viết là let-name, thì khi truyền data qua context, property của object truyền vào sẽ là $implicit. Ví dụ cụ thể:

<ng-template #buttonTmpl

let-label

let-className="className"

let-icon="icon">

<button [ngClass]="['btn', className ? className : '']">

<i \*ngIf="icon"

class="fa {{icon}}"></i>

{{ label }}

</button>

</ng-template>

<ng-container [ngTemplateOutlet]="buttonTmpl"

[ngTemplateOutletContext]="{ $implicit: 'Remove', className: 'btn-danger', icon: 'fa-remove' }">

</ng-container>

* Khi dùng variable ở trong ng-template, bạn sẽ mất type safe. Ví dụ bạn pass vào một object user theo cú pháp let-user="user" với firstName, lastName và age. Thì trong ng-template, bạn muốn làm gì cái object này cũng được, ví dụ như thử dùng user.fullName, compiler sẽ không catch được lỗi, và cả angular language service cũng không báo lỗi trên IDE.

## ng-container

ng-container là một custom html tag để khi render trên UI sẽ không có extra tag để tránh ảnh hưởng đến style mình viết. Như ở ví dụ trên, bạn hoàn toàn có thể viết lại thành.

<div

[ngTemplateOutlet]="buttonTmpl"

[ngTemplateOutletContext]="{ label: 'Click here', class: 'btn-primary', icon: null }">

</div>

Tuy nhiên khi render ra UI, sẽ bị thừa một thẻ div bao ngoài.

<div>

<button class="btn btn-primary">

Click here

</button>

</div>

Nếu bạn có style CSS chặt chẽ theo kiểu parent > child. Thì khi thừa một thẻ div, chắc chắn UI sẽ bị ảnh hưởng.

**- Basic Dependency Injection**

**Dependency Injection**

Dependency Injection (DI) là một pattern rất quan trọng trong lập trình, có thể bạn đã nghe qua những Spring Framework của Java hay ASP.NET có support DI, vậy với Frontend application thì có thật sự cần thiết?

Hiện nay có rất nhiều Frontend Framework có support DI, Angular sử dụng rất nhiều và nó có riêng một DI framework được thiết kế để đáp ứng được yêu cầu về mặt như kiến trúc hệ thống của Angular.

**DI là gì?**

Giả sử chúng ta đang làm ứng dụng giỏ hàng của một trang thương mại điện tử, thông thường chúng ta có thể tách những phần xử lý business ra thành một class để giảm responsibility của component, cũng như tăng tính reuseable của phần business logic đó.

Lúc này chúng ta có thể có các thành phần như sau:

class ProductModel {

sku: string;

name: string;

price: number;

}

interface CartItem {

product: ProductModel;

quantity: number;

}

class CartService {

selectedProducts: CartItem[] = [];

calculateTotal(): number {

return this.selectedProducts.reduce(

(total, item) => item.product.price \* item.quantity + total,

0

);

}

addToCart(): void {

// logic here

}

}

class ProductComponent {

cartService: CartService;

}

Các bạn có thể thấy rằng, class ProductComponent của chúng ta bị phụ thuộc vào (depend on) class CartService. Khi ProductComponent cần call các method của CartService chúng ta cần khởi tạo hoặc yêu cầu để lấy về một instance của service đó.

**Khởi tạo bên trong ProductComponent**

Để khởi tạo instance của một class bên trong một class khác, chúng ta có thể chỉ cần new là được.

class ProductComponent {

cartService: CartService;

constructor() {

this.cartService = new CartService();

}

}

// tương đương với

class ProductComponent {

cartService: CartService = new CartService();

}

Nếu trong trường hợp như trên, rõ ràng chúng ta bị ràng buộc chặt chẽ giữa hai class (Tight Coupling), nếu bạn muốn thay đổi implementation khác của CartService thì bắt buộc phải sửa code của ProductComponent, và sẽ kéo theo phải test lại cả hai class.

**Injection (Request để lấy về instance)**

Nếu bằng một cách nào đó, chúng ta gửi một request rồi từ một container nào đó sẽ cung cấp instance cho chúng ta thì sao.

class ProductComponent {

cartService: CartService;

constructor(cartService: CartService) {

this.cartService = cartService;

}

}

// tương đương cú pháp sau trong TypeScript

class ProductComponent {

constructor(public cartService: CartService) {}

}

Và chúng ta sẽ có một container có thể biết cách khởi tạo và provide cho các request của các class như sau:

(function container() {

const service = new CartService(); // và các dependencies của CartService nếu có

const productComp = new ProductComponent(service);

// other code logic

})();

Lúc này bạn có thể thấy class ProductComponent không hề biết gì về các khởi tạo CartService như thế nào. Nó gửi request cho Inversion of Control (đảo ngược sự điều khiển hay đảo ngược sự phụ thuộc, viết tắt là IoC) container, từ đó nó có thể lấy được instance mà nó mong muốn.

Trong trường hợp bạn muốn đổi implementation cũng rất dễ dàng và hoàn toàn không cần viết lại class ProductComponent.

Các bạn có thể đoán được mẫu thiết kế trên chính là Dependency Injection, chi tiết hơn đó là constructor injection.

**DI trong Angular**

Trong Angular, DI bao gồm ba thành phần sau đây:

* Injector: là một object có chứa các API để chúng ta có thể lấy về các instances đã tạo hoặc tạo các instances của các phụ thuộc.
* Provider: giống như một công thức để Injector có thể biết làm thế nào để tạo một instance của một phụ thuộc.
* Dependency: là một object (có thể là function, một value thông thường) của một kiểu dữ liệu cần phải khởi tạo.

Bạn có thể cung cấp injectors với providers ở nhiều levels khác nhau trong app, bằng một trong ba cách sau:

* Trong @Injectable() decorator cho service đó.
* Trong @NgModule() decorator (providers array) đối với NgModule.
* Trong @Component() decorator (providers array) đối với component hoặc directive (Chúng ta sẽ tìm hiểu chi tiết về Directive sau).

Ví dụ về cách provide CartService:

@Injectable({

providedIn: "root",

})

export class CartService {

// properties and methods

}

@Component({

selector: "app-product",

templateUrl: "./product.component.html",

styleUrls: ["./product.component.css"],

})

export class ProductComponent implements OnInit {

constructor(private cartService: CartService) {}

ngOnInit() {

console.log(this.cartService.calculateTotal());

}

}

Với @Injectable decorator, chúng ta đã thêm metadata để Angular biết được cách để tạo instance của CartService khi có một class nào đó request như ProductComponent. Và thông tin providedIn: 'root' sẽ chỉ cách cho Angular biết rằng chúng ta mong muốn service này sẽ có một instance (singleton) cho suốt toàn bộ app.

**Override Provider**

Giờ đây nếu hệ thống yêu cầu chúng ta thay vì lưu trữ tính toán thông tin giỏ hàng ở client, thì sẽ call đến một external datasource (API chẳng hạn) để thao tác như **Thêm vào giỏ hàng**, **Tính tổng tiền**. Lúc này về mặt coding, chúng ta sẽ tuân thủ theo đúng những gì class CartService đang có (public APIs của CartService), chỉ khác về mặt implementation như sau:

@Injectable()

export class CartExtService {

calculateTotal(): number {

// call external datasource

// return data from exteral datasource

return Math.random() \* 100;

}

addToCart(): void {

// logic here

}

}

Bạn chỉ cần có thế và có thể tiến hành override, mà không cần sửa đổi lại code của ProductComponent như sau:

@NgModule({

// other metadata

providers: [

{

provide: CartService,

useClass: CartExtService,

},

],

})

export class AppModule {}

Hoặc có thể override vào @Injectable của service

@Injectable({

providedIn: "root",

useClass: CartExtService,

})

export class CartService {

// logic here

}