

Mục lục

1.	Những tính năng nổi bật của Angular.....	3
2.	Cài đặt môi trường lập trình.....	3
3.	Cài đặt Angular CLI.....	3
4.	Tạo không gian làm việc và ứng dụng đầu tiên.....	3
5.	Các thành phần của Angular.....	3
6.	Khái niệm component.....	3
7.	Tạo component đơn giản.....	3
8.	Component decorators.....	5
9.	Nhúng bootstrap và jquery vào project.....	5
10.	String interpolation.....	6
11.	Tạo component sử dụng Angular CLI.....	7
12.	Property binding.....	7
13.	Event binding.....	8
14.	Xử lý text input.....	8
15.	Two way binding.....	9
16.	Template reference variable.....	9
17.	Style binding và class binding.....	9
18.	Cơ bản về directive, ngStyle.....	10
19.	ngIf.....	12
20.	ngFor đơn giản.....	12
21.	ngFor với mảng đối tượng.....	13
22.	Thêm phần tử vào mảng kết hợp ngFor.....	14
23.	Ẩn hiện form.....	15
24.	Xóa phần tử trong mảng hiển thị.....	15
25.	Lọc mảng hiển thị.....	16
26.	Thay đổi trạng thái phần tử trong mảng hiển thị.....	16
27.	Sử dụng Input để truyền tham số vào cho các component.....	16
28.	ngFor kết hợp Input.....	17

29.	<u>Output 1: setup.....</u>	18
30.	<u>Output 2: tiếp cận các thuộc tính và phương thức của Component cha từ Component con ..</u>	19
31.	<u>Ouput có tham số.....</u>	19
32.	<u>Bài tập Output.....</u>	21
33.	<u>Viewchild 1: setup.....</u>	22
34.	<u>ViewChild 2: tiếp cận các thuộc tính và phương thức của Component con từ Component cha... </u>	23
35.	<u>ngContent: tạo ra những component làm khung sườn cho những component khác.....</u>	23
36.	<u>Build in pipes.....</u>	24
37.	<u>Custom pipe.....</u>	26
38.	<u>Khái niệm service và http service.....</u>	27
39.	<u>Tách service.....</u>	28
40.	<u>Sử dụng ngOnInit.....</u>	29
41.	<u>Giới thiệu bài tập thời tiết.....</u>	29
42.	<u>API open weather map.....</u>	29
43.	<u>Weather service.....</u>	29
44.	<u>Lấy nhiệt độ và hiển thị.....</u>	30
45.	<u>Tính năng loading và xử lý lỗi.....</u>	30
46.	<u>Giới thiệu về form.....</u>	32
47.	<u>ngForm:.....</u>	32
48.	<u>Validate template form.....</u>	33
49.	<u>ngSubmit.....</u>	34
50.	<u>Xử lý style các input không hợp lệ.....</u>	34
51.	<u>Xử lý checkbox và ngModelGroup.....</u>	35
52.	<u>Reactive form.....</u>	37

ANGULAR

1. Những tính năng nổi bật của Angular:

- Tính năng tự động reload browser khi code thay đổi
- Giúp code có tính chặt chẽ cao hơn và ngắn gọn hơn
- Phiên bản Angular 4 là phiên bản mới, được nâng cấp từ phiên bản Angular 2
- Tài liệu tham khảo: <https://angular.io/>

2. Cài đặt môi trường lập trình

- Cài nodeJS: truy cập đường dẫn sau để download NodeJS phiên bản phù hợp với máy <https://nodejs.org/en/>

- Cài đặt Code editor: VS Code: <https://code.visualstudio.com/download>

3. Cài đặt Angular CLI

- Mở Terminal và gõ lệnh : **npm install -g @angular/cli**
- Kiểm tra xem đã cài đặt thành công chưa: **ng -v**

4. Tạo không gian làm việc và ứng dụng đầu tiên

- Tạo project mới: **ng new ProjectName**
- Chạy project lên: **ng serve --open**
- Mở trình duyệt với địa chỉ: <http://localhost:4200/> để xem kết quả

5. Các thành phần của Angular

- ./src/app/app.component.ts.
- ./src/app/app.component.css
- ./src/app/app.component.html
- ./src/app/app.module.ts
- ./src/index.html

6. Khái niệm component

a. Component là gì?

- Component là các thành phần được hiển thị trên màn hình
- Điều chỉnh một phần của màn hình (view)
- Component có thể chứa nhiều thẻ HTML
- Component có thể chứa Component khác
- Component có thể nhận tham số
- Component có thể rất linh hoạt

b. Lợi ích của việc sử dụng Component

- Dễ quản lý code
- Tái sử dụng code

7. Tạo component đơn giản

- Việc viết ra một component là tạo ra một HTML tag, để có thể sử dụng trong các component khác
- Quy tắc đặt tên của component là **tên_component.component.ts**
- Ví dụ:

1. Trong thư mục ./src/app/ tạo ra file **word.component.ts**

2. Trong file **word.component.ts**, import các thư viện cần thiết vào và tạo ra các thành phần cần thiết cho một component như sau:

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

@Component({
  selector: 'app-word',
  template: '<h3 style="color: blue">This is Word Component</h3>',
})

export class WordComponent {}
```

3. Import component này vào file **app.module.ts**

```
import { WordComponent } from './word.component';
```

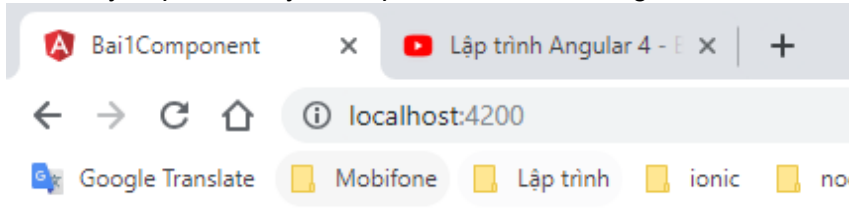
4. Khai báo component này trong mảng **declarations** của file **app.module.ts**, như sau:

```
@NgModule({
  declarations: [
    AppComponent,
    WordComponent,
  ],
```

5. Sử dụng component vừa tạo trong file **app.component.html**, như sau:

```
<h1>This is app component</h1>
<app-word></app-word>
```

6. Truy cập trình duyệt với port 4200, ta được giao diện như sau:



This is app component

This is Word Component

8. Component decorators

- Có thể tách riêng các template, style ra thành các file riêng sau đó link vào trong file ts để sử dụng, như vậy code sẽ gọn hơn

a) Tạo các file **word.component.css**, **word.component.html** để định nghĩa các thành phần này trong các file riêng biệt

- File **word.component.html**:

```
<> word.component.html x
1  <h3>This is Word Component</h3>
```

- File **word.component.css**

```
# word.component.css x
1  h3 {
2    color: blue;
3  }
```

a) Khai báo các file này vào file **word.component.ts** để sử dụng

```
TS word.component.ts x
1  import { Component } from '@angular/core'
2
3  @Component({
4    selector: 'app-word',
5    templateUrl: './word.component.html',
6    styleUrls: ['./word.component.css']
7  })
8
9  export class WordComponent {}
```

9. Nhúng bootstrap và jquery vào project

- Cài bootstrap và jquery: **npm i bootstrap jquery --save**

- Thêm đường dẫn **bootstrap.min.css** vào file **angular.json**

- Thêm đường dẫn **bootstrap.min.js** và **jquery.min.js** vào file **angular.json**

```
"styles": [
  "src/styles.css",
  "node_modules/bootstrap/dist/css/bootstrap.min.css"
],
"scripts": [
  "node_modules/bootstrap/dist/js/bootstrap.min.js",
  "node_modules/jquery/dist/jquery.min.js"
]
```

10. String interpolation

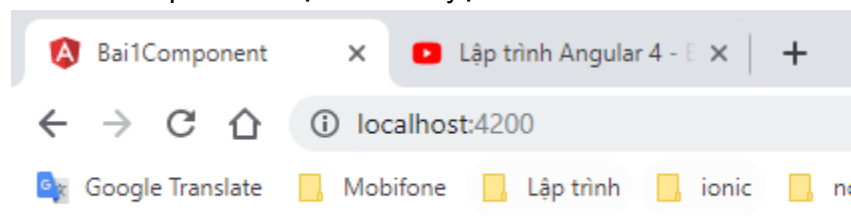
- Truyền giá trị từ class sang file HTML để hiển thị ra trình duyệt
- Trong file **word.component.ts**, tạo các biến **en** và **vn** như sau:

```
export class WordComponent {  
  en: string = "Hello";  
  vn: string = "Xin chào";  
}
```

- Trong file **word.component.html**, dùng string interpolation để hiển thị giá trị các biến được định nghĩa ở file **word.component.ts**, như sau:

```
<h3>This is Word Component</h3>  
<h4>{{ en }}</h4>  
<p>{{ vn }}</p>
```

- Kết quả hiển thị ở trình duyệt



This is app component

This is Word Component

Hello

Xin chào

Nhận xét: Nếu để các file chung ngoài thư mục **.src/app/** thì sẽ có rất nhiều file lộn xộn, để cho cấu trúc project gọn gàng và dễ quản lý, ta tạo thư mục **word** rồi di chuyển các file của component word vào

trong thư mục này, sau đó sửa đường dẫn import trong **app.module.ts** cho đúng.

11. Tạo component sử dụng Angular CLI

- Dùng lệnh Angular CLI để tạo các component: **ng g c tên_component**
- Một folder mới được tạo ra chứa 4 file của component mới là **css**, **html**, **spec.ts** và **ts**
- File **.spec.ts** là file dùng để test component
- Ngoài ra thì component vừa tạo này cũng tự động import vào file **app.module.ts**
- Lúc mới tạo component thì Angular cũng tự động implement một interface **OnInit** vào, vì vậy trong class phải có hàm **ngOnInit()**, hàm này sẽ tự động chạy khi gọi component, khi component thay đổi trạng thái, khi component khởi tạo, có thể dùng hàm này để lấy dữ liệu từ server, chi tiết xem thêm tại document

12. Property binding

- Tạo các biến trong class **WordComponent**, trong đó biến **imageUrl** là đường dẫn của một hình ảnh, như sau:

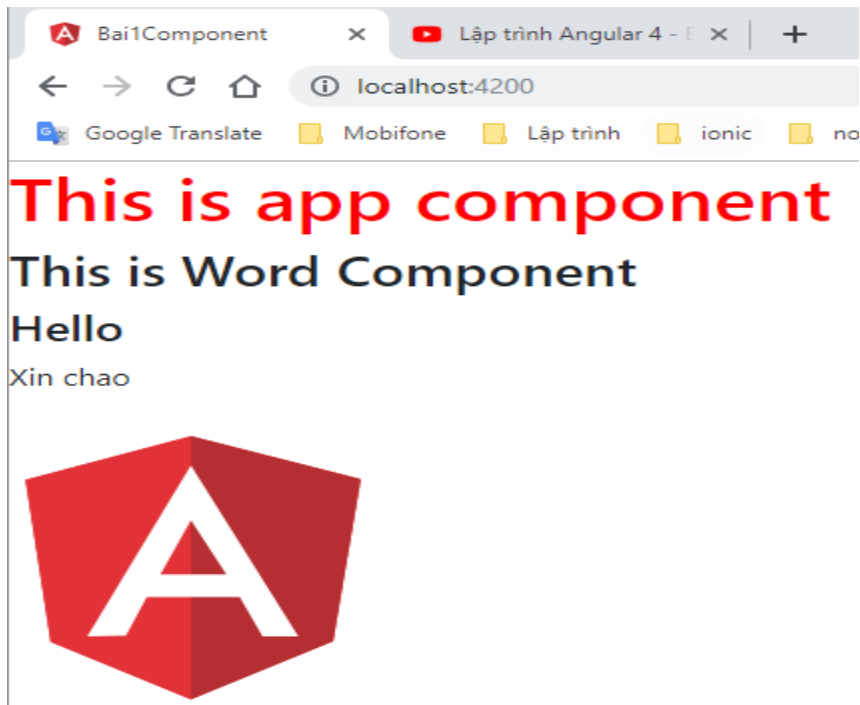
```
export class WordComponent implements OnInit {  
  en = "Hello";  
  vn = "Xin chào";  
  imageUrl = "https://angular.io/assets/images/logos/angular/shield-large.svg";  
}
```

- Sử dụng các biến này trong file html, trong đó, thuộc tính **src** để trong dấu ngoặc vuông, còn giá trị **imageUrl** được để trong dấu ngoặc kép, khi đó Angular sẽ hiểu thuộc tính **src** của thẻ **img** sẽ có giá trị là giá trị của biến **imageUrl**:

(thuộc tính **src** sử dụng property binding)

```
<h3>This is Word Component</h3>  
<h4>{{ en }}</h4>  
<p>{{ vn }}</p>  
<img [src]="imageUrl" alt="">
```

- Kết quả hiển thị ra trình duyệt



- Thêm biến **forgot** = “true” vào class **WordComponent**
- Sử dụng property binding biến **forgot** trong file html, như sau:

```
<h4 [hidden]="forgot">{{ en }}</h4>
```

- Khi đó thẻ **<h4>** sẽ bị ẩn đi, khi biến **forgot** có giá trị **true**

13. Event binding

- Qua những bài học vừa rồi, ta đã sử dụng **string interpolation**, **property binding** để gán giá trị của biến từ file ts sang file html. Bây giờ, dùng event binding để gán giá trị từ file html sang file ts

- Thêm một thẻ button vào file html, tạo event binding cho sự kiện click, để khi kích vào thì phủ định lại giá trị của biến **forgot**, như sau:

```
<button (click)="forgot = !forgot">Toggle forgot</button>
```

- Thông thường ta không gán event binding với một phép toán như trên mà gán với một hàm

```
<button (click)="toggleForgot()">Toggle forgot</button>
```

- Định nghĩa hàm này trong file ts:

```
toggleForgot(){  
  | this.forgot = !this.forgot;  
}
```

14. Xử lý text input

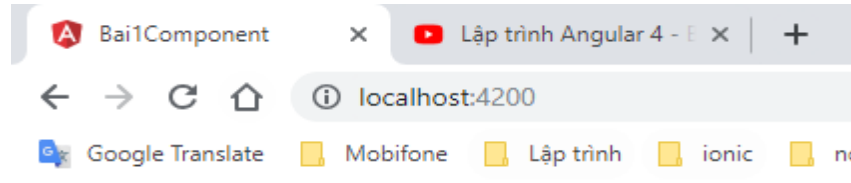
- Trong phần này, ta sẽ sử dụng kết hợp **property binding** và **event binding**
- Dùng Angular CLI, tạo một component mới tên là user-form: **ng g c user-form**
- ở file html, tạo một thẻ input để nhập dữ liệu và có một thẻ h3 để hiển thị dữ liệu, như sau:

```
<input placeholder="Enter your username" (keyup)="showEvent($event)">  
<h3>Your name is {{ name }}</h3>
```

- Mỗi khi nhập dữ liệu vào ô input thì sự kiện **keyup** sẽ thực hiện hàm **showEvent(\$event)**, định nghĩa hàm **showEvent(\$event)** trong file ts, như sau:


```
showEvent(event){
  this.name = event.target.value;
}
```

- Tham số **\$event** là biến chứa nhiều thông tin về event mà mình vừa làm, biến này chứa nhiều thuộc tính, sẽ được truy cập để lấy các giá trị cần thiết
- Khi chạy chương trình, mỗi khi nhập dữ liệu vào ô input thì giá trị vừa nhập sẽ được gán cho biến **name**, và được hiển thị trong thẻ h3, như sau:



This is app component

Your name is Angular

15. Two way binding

- Để sử dụng được two way binding, cần import vào **app.module.ts** như sau:

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

- Thêm vào mảng import

```
imports: [
  BrowserModule,
  AppRoutingModule,
  FormsModule
],
```

- Thêm thuộc tính **[(ngModel)]** vào thẻ input như sau:

```
<input placeholder="Enter your username" [(ngModel)]="name">
```

- Như vậy thì biến name sẽ được binding 2 chiều, từ file ts sang file html và ngược lại

16. Template reference variable

- Chỉnh sửa lại file html, thêm biến template reference, và hiển thị ra như sau, nhớ phải có thuộc tính **ngModel**

```
<input placeholder="Enter your username" ngModel #txtUsername>
<h3>Your name is {{ txtUsername.value | json }}</h3>
```

17. Style binding và class binding

- Thêm thuộc tính **style.color** là "red" vào thẻ h3 để đoạn text trong này hiển thị màu đỏ theo ý thích, như sau:

```
<h3 [style.color]="red">Your name is {{ name }}</h3>
```

- Có thể thêm biểu thức vào cho thuộc tính **style.color** để khi chiều dài giá trị của biến name là chẵn thì hiển thị chữ màu đỏ, ngược lại thì hiển thị chữ màu đen, như sau:

```
<h3 [style.color]="name.length%2==0? 'red': 'black'">Your name is {{ name }}</h3>
```

- Trong file css, tạo ra các định dạng css cho các biến circle và square như sau:

```
.circle{
  width: 50px;
  height: 50px;
  border-radius: 25px;
  background-color: blue;
}
.square{
  width: 50px;
  height: 50px;
  border-radius: 2px;
  background-color: red;
}
```

- Trong file html, sử dụng class binding để khi giá trị biến isHighLight là true thì áp dụng class **circle**, ngược lại thì áp dụng class **square**

```
<div [class]="isHighLight?'circle':'square'"></div>
```

(nếu dùng binding theo cách này thì giá trị ở trong dấu "" là một biểu thức hoặc giá trị của một biến)

- Có thể dùng class binding theo cách này:

```
<div [class.circle]="isHighLight" [class.square]="!isHighLight"></div>
```

18. Cơ bản về directive, ngStyle

- Có 3 kiểu directive trong Angular:
 1. Component: directive với 1 Template
 2. Attribute directive: thay đổi việc hiển thị hoặc hành vi của các DOM, component hoặc directive khác
 3. Structural directive: thường thêm bớt các DOM (**ngIf**, **ngFor**)
- Dùng **[ngStyle]** để định nghĩa nhiều thuộc tính style cho một tag

```
<h3 [ngStyle] = "{ color: 'red', fontSize: '20px' }">Your name is {{ name }}</h3>
```

- Có thể định nghĩa biến **currentStyle** ở file ts, để gán các thuộc tính cho biến

```
currentStyle = { color: 'red', fontSize: '20px' };
```

- Sau đó sử dụng biến này ở file html

```
<h3 [ngStyle] = "currentStyle">Your name is {{ name }}</h3>
```

- Có thể định nghĩa nhiều biến chứa các giá trị khác nhau của style

```
evenStyle = { color: 'red', fontSize: '20px' };
oddStyle = { color: 'black', fontSize: '40px' };

```

- Sau đó sử dụng các biến này ở file html với các điều kiện khác nhau:

```
<h3 [ngStyle]="name.length%2==0? evenStyle : oddStyle">Your name is {{ name }}</h3>
```

- Thuộc tính **ngClass** cũng tương tự như thuộc tính **ngStyle**
- Giá trị truyền vào của các thuộc tính **ngStyle** và **ngClass** là một **Object**

```
<div [ngClass]="{ circle : isHighLight, square : !isHighLight }"></div>
```

Ví dụ: Xây dựng một **Attribute directive** đơn giản

Xây dựng một **Attribute directive** để đặt màu nền cho một phần tử khi người dùng di chuyển chuột qua nó

1. Khởi tạo lớp **directive** mới bằng lệnh Angular CLI: **ng g d highlight**
2. Import thư viện **ElementRef**
3. Chỉnh sửa lại file **highlight.directive.ts** như sau:

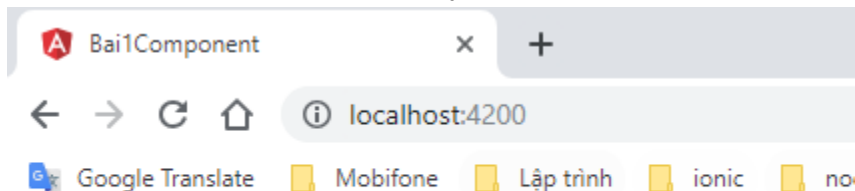
```
import { Directive, ElementRef } from '@angular/core';

@Directive({
  selector: '[appHighlight]'
})
export class HighlightDirective {
  constructor(el: ElementRef) {
    el.nativeElement.style.backgroundColor = 'yellow';
  }
}
```

4. Để sử dụng directive vừa tạo ra, thêm nó vào một thành phần trong file html

```
<h1 appHighlight>This is app component</h1>
```

5. Kết quả hiển thị ra trình duyệt như sau:



This is app component

Tiếp theo, làm thêm để khi người dùng di chuyển chuột qua thành phần thì nó sẽ có hiệu ứng

6. Import thêm thư viện **HostListener** vào

```
import { Directive, ElementRef, HostListener } from '@angular/core';
```

7. Sau đó, thêm hai trình xử lý sự kiện phản hồi khi người dùng di chuyển chuột vào và rời chuột khỏi

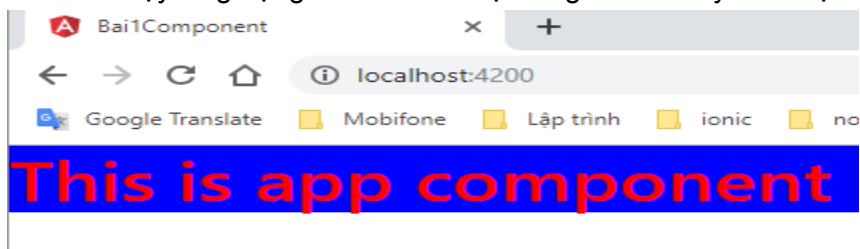
```
private highlight(color: string) {
  this.el.nativeElement.style.backgroundColor = color;
}
@HostListener('mouseenter') onMouseEnter() {
  this.highlight("blue");
}

@HostListener('mouseleave') onMouseLeave() {
  this.highlight(null);
}
```

8. Thêm biến ElementRef vào constructor

```
constructor(private el: ElementRef) {}
```

9. Chạy ứng dụng lên để xem hiệu ứng khi di chuyển chuột vào thì sẽ như sau:



19. ngIf

- *ngIf là một **Structural directive**
- Cú pháp đơn giản của *ngIf

```
<div *ngIf="condition">Content to render when condition is true.</div>
```

- *ngIf với else:

```
<div *ngIf="condition; else elseBlock">
  Content to render when condition is true.
</div>
<ng-template #elseBlock>
  Content to render when condition is false.
</ng-template>
```

20. ngFor đơn giản

- *ngFor để lặp một dãy các giá trị và hiển thị ra
- Khai báo một mảng các môn học trong file ts:

```
arrSubjects = ['Angular', 'NodeJS', 'React'];
```

- Dùng *ngFor để hiển thị ra trong file html

```
<ul *ngFor="let subject of arrSubjects">
  <li>{{ subject }}</li>
</ul>
```

- Kết quả hiển thị ở trình duyệt

- Angular
- NodeJS
- React

21. **ngFor với mảng đối tượng**

- Dùng ***ngFor** để lặp các phần tử trong một mảng các **object**
- Tạo một component mới có tên **words**
- Trong file ts, tạo một mảng các **words** như sau:

```
arWords = [
  { id: 1, en: 'action', vn: 'hành động', memorized: true },
  { id: 2, en: 'actor', vn: 'diễn viên', memorized: false },
  { id: 3, en: 'activity', vn: 'hoạt động', memorized: true },
  { id: 4, en: 'active', vn: 'chủ động', memorized: true },
  { id: 5, en: 'bath', vn: 'tắm', memorized: false },
  { id: 6, en: 'bedroom', vn: 'phòng ngủ', memorized: true }
];
```

- Trong file html, dùng lệnh ***ngFor** để lặp các phần tử trong mảng và hiển thị ra như sau:

```
<div *ngFor="let word of arWords">
  <h4>{{ word. en }}</h4>
  <p>{{ word. vn }}</p>
</div>
```

- Thêm thuộc tính **[ngStyle]** để định dạng các thẻ h4 có màu xanh khi thuộc tính **memorized** là **true** và màu đỏ khi thuộc tính **memorized** là **false**

```
<h4 [ngStyle]="{color: word.memorized? 'green': 'red'}">
  {{ word. en }}
</h4>
```

- Kết quả hiển thị ra trình duyệt như sau:

action

hành động

actor

diễn viên

activity

hoạt động

active

chủ động

bath

tắm

bedroom

phòng ngủ

22. Thêm phần tử vào mảng kết hợp ngFor

- Tạo các thẻ input để nhập các dữ liệu ở file html
- Tạo button để khi kích vào thì thực hiện hàm **addWord()**

```
<div>
  <input placeholder="English" [(ngModel)]="en"><br />
  <input placeholder="Tiếng Việt" [(ngModel)]="vn"><br />
  <button (click)="addWord()">Thêm</button>
</div>
```

- Khai báo các biến **en** và **vn** để lưu các giá trị nhập vào từ file html

```
vn: string;
```

```
en: string;
```

- Định nghĩa hàm **addWord()** thực hiện thêm từ mới vào mảng **arWords**

```
addWord() {
  let word = { id: this.arWords.length + 1, en: this.en, vn: this.vn, memorized: false };
  this.arWords.unshift(word);
}
```

- Kết quả hiển thị ra trình duyệt

table
cái bàn

Thêm

table

cái bàn

23. Ẩn hiện form

- **Mô tả:**

1. Lúc mới khởi động trình duyệt lên thì không hiện form thêm từ mới, chỉ hiện danh sách các từ và button **thêm từ mới**
2. Khi kích vào button **thêm từ mới** thì hiện form để thêm từ mới
3. Sau khi thêm từ mới xong thì ẩn form đi

- **Các bước thực hiện:**

1. Tạo button thêm từ mới: `<button>Thêm từ mới</button>`
2. Tạo biến **isShow=false** để ban đầu thì không hiện form `isShow = false;`
3. Sử dụng ***ngIf** để ẩn form đi

```
<div *ngIf="isShow">
  <input placeholder="English" [(ngModel)]="en"><br />
  <input placeholder="Tiếng Việt" [(ngModel)]="vn"><br />
  <button (click)="addWord()">Thêm</button>
</div>
```

4. Trong định nghĩa hàm **addWord()**, sau khi thêm từ mới xong thì thay đổi giá trị biến

isShow để ẩn form đi `this.isShow = !this.isShow;`

24. Xóa phần tử trong mảng hiển thị

- Thêm định dạng trong file css, để tách riêng mỗi phần tử hiển thị cho đẹp
- Với mỗi phần tử, thêm một button để khi kích vào thì thực hiện hàm **deleteWord** xóa phần tử đó khỏi mảng, tham số truyền vào là **id** của phần tử muốn xóa

```
<button (click)="deleteWord(word.id)">Xóa</button>
```

- Định nghĩa hàm **deleteWord**, như sau:

```
deleteWord(id) {
  let index = this.arWords.findIndex(e => e.id==id);
  this.arWords.splice(index, 1);
}
```

25. Lọc mảng hiển thị

1. Thêm thẻ **select** để xác định các từ muốn lọc

```
<select [(ngModel)]="filterStatus">
  <option value="tat_ca">Tất cả</option>
  <option value="da_nho">Đã nhớ</option>
  <option value="chua_nho">Chưa nhớ</option>
</select>
```

2. Dùng ***ngIf** để hiển thị những từ muốn lọc dựa vào **filterStatus** và thuộc tính **memorized** của từ

```
<div *ngFor="let word of arWords">
  <div *ngIf="filterStatus=='tat_ca' || filterStatus=='da_nho' && word.memorized || filterStatus=='chua_nho' && !word.memorized">
    <h4 [ngStyle]="{color: word.memorized? 'green': 'red'}">
      {{ word.en }}
    </h4>
    <p>{{ word.vn }}</p>
  </div>
</div>
```

3. Có thể tách riêng câu lệnh điều kiện ở ***ngIf** vào hàm **getShowStatus** cho gọn như sau:

```
<div *ngIf="getShowStatus(word.memorized)">
```

- Định nghĩa hàm **getShowStatus** ở file ts

```
getShowStatus(memorized) {
  let xemTatCa = this.filterStatus == 'tat_ca';
  let xemDaNho = this.filterStatus == 'da_nho' && memorized;
  let xemChuaNho = this.filterStatus == 'chua_nho' && !memorized;
  return xemTatCa || xemDaNho || xemChuaNho;
}
```

26. Thay đổi trạng thái phản từ trong mảng hiển thị

- Tạo button để khi kích vào thì thay đổi thuộc tính của đối tượng trong mảng hiển thị

```
<button (click)="word.memorized = !word.memorized">{{ word.memorized? "Chưa thuộc": "Đã thuộc"}}</button>
```

27. Sử dụng Input để truyền tham số vào cho các component

- Tạo ra các component có thể truyền tham số vào khi sử dụng, như vậy component sẽ linh hoạt hơn, và có tính tái sử dụng
- Tạo component mới có tên **person**
- Trong file html, sử dụng các thẻ h3 và p để hiển thị tên và tuổi

```
<h3>Teo Nguyen</h3>
<p>18</p>
```

- Sử dụng component **person** trong file **app.component.html**

```
<h1>This is app component</h1>
<app-person></app-person>
```

- Kết quả hiển thị ra trình duyệt

Teo Nguyen

- Nhận thấy, component **person** chỉ được sử dụng với giá trị gán cứng và không thể gán giá trị khác cho component này được. Bây giờ sử dụng **Input** để gán giá trị cho component, như vậy component sẽ linh hoạt hơn, có thể tái sử dụng được
- Trong file **app.component.html**, truyền thêm các biến vào cho component **person**:

```
<h1>This is app component</h1>
<app-person name="Teo Nguyen" age="18"></app-person>
```

- Trong file **person.component.ts**, import thêm thư viện **Input**

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

- Cũng trong file này, nhận các biến được truyền vào từ component

```
export class PersonComponent implements OnInit {
  @Input() name: String;
  @Input() age: String;
```

- Trong file **person.component.html**, dùng string interpolation để hiển thị giá trị các biến ra

```
<h3>{{ name }}</h3>
<p>{{ age }}</p>
```

- Kết quả hiển thị ra trình duyệt cũng tương tự như trên, nhưng bây giờ, component **person** đã có thể tái sử dụng bằng cách truyền tham số vào cho nó:

```
<app-person name="Teo Nguyen" age="18"></app-person>
<app-person name="Ti Nguyen" age="15"></app-person>
```

- Kết quả hiển thị ra trình duyệt

```
Teo Nguyen
18
Ti Nguyen
15
```

28. ngFor kết hợp Input

- Sử dụng ***ngFor** để lặp mảng đối tượng và hiển thị ra, dùng **input** để truyền tham số vào cho component
- Tạo component mới **list-person**
- Trong file ts, khai báo mảng **arPeople** như sau:

```
arPeople = [
  { name: "Teo", age: 18 },
  { name: "Ti", age: 15 },
  { name: "Tun", age: 16 }
]
```

- Trong file html, dùng ***ngFor** để lặp mảng **arPeople** (truyền tham số vào cho các biến) và hiển thị ra như sau:

```
<div *ngFor="let person of arPeople">
  <app-person [name]="person.name" [age]="person.age"></app-person>
</div>
```

Chú ý:

- các tham số **name**, **age** phải dùng property binding hoặc string interpolation
- Trong lệnh ***ngFor** ở trên, nếu để câu lệnh ***ngFor** trong thẻ div thì sẽ hình thành một thẻ div bao component **person** lại (bị dư ra một thẻ div),

```
<div _ngcontent-c2>
  <app-person _ngcontent-c2 _ngghost-c1 ng-reflect-name="Ti" ng-reflect-age="15">...</app-person>
</div>
```

- để tránh điều này, sử dụng **<ng-container>** thay cho thẻ div, như sau:

```
<ng-container *ngFor="let person of arPeople">
  <app-person [name]="person.name" [age]="person.age"></app-person>
</ng-container>
```

Lúc này thì không bị dư thẻ div nữa

```
<app-person _ngcontent-c2 _ngghost-c1 ng-reflect-name="Ti" ng-reflect-age="15">
...</app-person>
<!-->
```

29. Output 1: setup

- Mô tả: Setup để **parent-component** hiển thị **value** của nó và render **child component** vào
- Các bước thực hiện:

1. Tạo **parent-component.ts** như sau, nhớ import vào **app.module.ts**:

```
import { Component } from "@angular/core"

@Component({
  selector: 'app-parent',
  template: `
    <h2>{{ value }}</h2>
    <app-child></app-child>
  `,
})

export class ParentComponent {
  value = 0;
}
```

2. Tạo **child-component.ts** như sau, nhớ import vào **app.module.ts**:

```
import { Component } from "@angular/core"

@Component({
  selector: "app-child",
  template: `
    <button>ADD</button>
  `,
})

export class ChildComponent {}
```

3. Sử dụng **parent** component trong file **app.component.html**

```
<h1>This is app component</h1>
<app-parrent></app-parrent>
```

4. Kết quả hiển thị ra trình duyệt:

0

ADD

30. Output 2: tiếp cận các thuộc tính và phương thức của Component cha từ Component con

- Các bước thực hiện:

1. Trong **parent.component.ts**, khai báo một sự kiện **myClick** ở component **app-child**, sự kiện này do mình tự đặt tên

```
<app-child (myClick)="value = value + 1"></app-child>
```

2. Trong **child.component.ts**:

a) Import thêm thư viện **Output** và **EventEmitter** vào

```
import { Component, Output, EventEmitter } from "@angular/core"
```

b) Nhận sự kiện **myClick**:

```
export class ChildComponent{
  @Output() myClick = new EventEmitter();
```

c) Thêm sự kiện **click** vào button của **child.component**

```
<button (click)="addForParrent()">ADD</button>
```

d) Định nghĩa hàm **addForParrent()** để khi kích vào button thì phát sinh sự kiện **myClick**

```
addForParrent(){
  this.myClick.emit();
}
```

3. Kết quả hiển thị ở trình duyệt, mỗi khi kích vào button **ADD** thì phát sinh sự kiện **myClick** để tăng giá trị **value** lên 1

1

ADD

31. Output có tham số

- Làm thế nào để truyền tham số vào cho các sự kiện **Output**?

- Các bước thực hiện:

+ Sử dụng hàm

1. Thay đổi sự kiện **myClick** ở parent component là một hàm

```
<app-child (myClick)="changeValue()"></app-child>
```

changeValue 2. Định nghĩa hàm **changeValue** ở trong class parent

component

```
export class ParrentComponent {
  value = 0;
  changeValue(){
    this.value = this.value + 1;
  }
}
```

- Kết quả hiển thị ra trình duyệt cũng tương tự như trên

+ truyền tham số (1)

1. Truyền tham số vào cho hàm `changeValue` là **true** (hoặc **false**)

```
<app-child (myClick)="changeValue(true)"></app-child>
```

2. Định nghĩa hàm `changeValue` để khi tham số là **true** thì tăng giá trị biến `value` lên 1, ngược lại thì giảm đi 1

```
changeValue(isAdd: boolean){
  if(isAdd){
    this.value = this.value + 1;
  } else {
    this.value = this.value - 1;
  }
}
```

3. Kết quả hiển thị ra trình duyệt, mỗi khi kích vào button thì phát sinh sự kiện **myClick** để thực hiện hàm **changeValue**, và tăng giá trị biến **value** lên 1, vì truyền vào tham số là **true**, (nếu truyền vào tham số là **false** thì sẽ giảm **value** đi 1)

1

ADD

4. Có thể truyền vào hàm `changeValue` tham số là **\$event**

```
<app-child (myClick)="changeValue($event)"></app-child>
```

5. Ở file **child.component.ts**, truyền giá trị **true** (hoặc **false**) vào sự kiện `emit`

```
addForParrent(){
  this.myClick.emit(true);
}
```

6. Kết quả hiển thị ra trình duyệt cũng tương tự như trên

+ truyền tham số (2)

1. Ở file **child.component.ts**, tạo thêm một button để khi kích vào thì thực hiện hàm **subForParrent**

```
<button (click)="addForParrent()">ADD</button>
<button (click)="subForParrent()">SUB</button>
```

2. Định nghĩa hàm **subForParrent**, truyền giá trị **false** vào sự kiện `emit`

3. Kết quả hiển thị ra ở trình duyệt, khi nhấn button **ADD** thì sẽ thực hiện hàm **changeValue**, tham số nhận vào là **true**, do hàm **addForParrent** ở **child.component** emit sang, còn nếu nhấn button **SUB** thì ngược lại

*Lưu ý: Nếu chỉ muốn truyền tham số kiểu boolean thì phải thêm ràng buộc này vào EventEmitter

```
@Output() myClick = new EventEmitter<boolean>();
```

32. Bài tập Output

- **Mô tả:** sử dụng lại component person và list-person, tạo thêm nút xóa ở mỗi person, khi kích vào nút nào thì sẽ xóa phần tử đó ra khỏi mảng arPeople
- **Các bước thực hiện:**
 1. ở file list-person.component.html, thêm sự kiện removePerson ở trong thẻ app-person, để khi kích vào nút này thì thực hiện hàm removePersonByName

```
<app-person [id]="person.id" [name]="person.name" [age]="person.age" (removePerson) = "removePersonByName($event)"></app-person>
```

2. trong file list-person.component.ts, định nghĩa hàm removePersonByName tham số truyền vào là name, thực hiện xóa phần tử trong mảng arPeople dựa vào name

```
removePersonByName(name){  
  let index = this.arPeople.findIndex(e => e.name == name);  
  this.arPeople.splice(index, 1);  
}
```

3. trong file person.component.ts, nhận sự kiện removePerson

```
@Output() removePerson = new EventEmitter<string>();
```

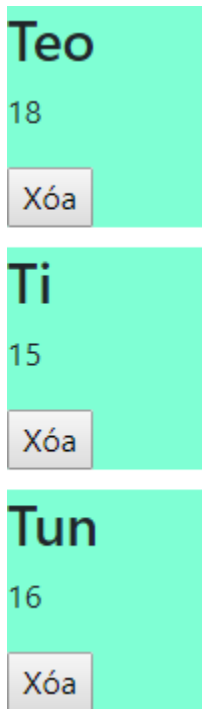
4. Trong file person.component.html, tạo một button Xóa, khi kích vào thì chạy hàm removeByClick để phát sinh sự kiện removePerson

```
<button (click)="removeByClick()">Xóa</button>
```

5. Trong file person.component.ts, định nghĩa hàm removeByClick để phát sinh sự kiện removePerson, tham số truyền vào là biến name từ input

```
removeByClick(){  
  this.removePerson.emit(this.name);  
}
```

6. Kết quả hiển thị ra trình duyệt, khi kích vào button Xóa thì xóa phần tử đó ra khỏi mảng



33. Viewchild 1: setup

- Mô tả:
 1. tạo child component, có thẻ h3 hiển thị giá trị value
 2. tạo parent component, có button add for child, và render child component
- Các bước thực hiện
 1. Tạo file child.component.ts, có biến value và hiển thị giá trị của biến trong thẻ h3

```
import { Component, Output, EventEmitter } from "@angular/core"

@Component({
  selector: "app-child",
  template: `
    <h3>{{ value }}</h3>
  `,
})

export class ChildComponent {
  value = 0;
}
```

2. Tạo file parent.component.ts, có button add for child và render child component vào

```
import { Component } from "@angular/core"

@Component({
  selector: 'app-parent',
  template: `
    <button>Add for child</button>
    <app-child></app-child>
  `,
})

export class ParrentComponent {}
```

34. ViewChild 2: tiếp cận các thuộc tính và phương thức của Component con từ Component cha

- **Mô tả:** Làm thế nào để khi kích vào button ở parent component thì có thể thay đổi được giá trị của child component
- **Các bước thực hiện:**

*cách 1: Khai báo biến variable ở trong app-child, khi đó có thể dùng biến này để lấy các thuộc tính và phương thức của child component, vì vậy có thể dùng biến này để tăng giá trị value lên 1

```
<button (click)="child.value = child.value + 1">Add for child</button>
<app-child #child></app-child>
```

Kết quả hiển thị ra trình duyệt, mỗi khi kích vào button thì sẽ tăng giá trị lên 1

Add for child

1

*cách 2:

1. trong parent.component.ts, import thư viện ViewChild, và ChildComponent vào

```
import { Component, ViewChild } from "@angular/core"
import { ChildComponent } from '../child.component'
```

2. Nhận component ViewChild và khởi tạo biến myChild thuộc lớp ChildComponent

```
@ViewChild(ChildComponent)
myChild: ChildComponent;
```

3. Tại templates, khi kích vào button thì thực hiện hàm onAddForChild

```
<button (click)="onAddForChild()">Add for child</button>
<app-child></app-child>
```

4. Tại class, định nghĩa hàm onAddForChild để tăng giá trị biến value lên 1

```
onAddForChild(){
  this.myChild.value++;
}
```

5. Kết quả hiển ra ở trình duyệt cũng tương tự như trên

35. ngContent: tạo ra những component làm khung sườn cho những component khác

- Tạo component card.component.ts hiển thị một thẻ card, như sau:

```
<div class="card">
  <p>Khoa Pham Training</p>
</div>
```

- Định nghĩa định dạng CSS cho class **card**
- Kết quả hiển thị ra trình duyệt là một card với nội dung được gán cứng

Khoa Pham Training

- Bây giờ, không truyền nội dung gán cứng cho component **card** nữa mà truyền cho nó một thẻ **<ng-content>**

```
<div class="card">
  <ng-content></ng-content>
</div>
```

- Sử dụng component này trong file **app.component.html**, truyền nội dung cho **card** component, như vậy việc sử dụng component **card** sẽ linh động hơn, có thể tái sử dụng các component

```
<app-card>
  <h4>Khoa Pham Training</h4>
</app-card>
<app-card>
  <p>Khoa Pham Training</p>
</app-card>
```

- Kết quả hiển thị ra trình duyệt

Khoa Pham Training

Khoa Pham Training

- Tiếp theo, truyền cho các thẻ **ng-content** thuộc tính **select** là class được khai báo ở **app.component.html**

```
<div class="card">
  <ng-content select=".card-header"></ng-content>
  <ng-content select=".card-body"></ng-content>
</div>
```

- Sử dụng component này trong file **app.component.html**, nhớ định nghĩa các class

```
<app-card>
  <h4 class="card-header">Khoa Pham Training</h4>
  <p class="card-body">Khoa Pham Training</p>
</app-card>
```

***Lưu ý:** Thứ tự hiển thị các thẻ phụ thuộc vào thứ tự ở trong component **card**

36. Build in pipes

- Pipe là công cụ để định dạng lại dữ liệu để show ra trình duyệt cho người dùng xem
- Hầu hết các ứng dụng đều làm 3 công việc như sau:

1. Lấy dữ liệu (từ server hoặc giá trị các biến trong component)
2. Chuyển đổi dữ liệu đó thành dạng thân thiện với người dùng để người dùng có thể đọc hiểu được
3. Show dữ liệu cho người dùng xem

- Ví dụ:

1. Tạo component mới đặt tên là **learn-pipe**
2. Trong file ts, tạo biến **birthday** có kiểu dữ liệu **Date**

```
birthday = new Date("15 Aug 2015");
```

3. Trong file html, hiển thị giá trị biến **birthday**

```
<p>
  {{ birthday }}
</p>
```

4. Kết quả hiển thị ra trình duyệt là dữ liệu ở dạng thô

Sat Aug 15 2015 00:00:00 GMT+0700 (Giờ Đông Dương)

- Bây giờ dùng công cụ Pipe để chuyển đổi dữ liệu từ dạng thô sang dạng thân thiện hơn để người dùng có thể đọc hiểu được
- Truy cập vào <https://angular.io/api?query=pipe> để xem các Pipe được tạo sẵn của Angular và sử dụng

1. Chỉnh sửa lại file html, sử dụng pipe để chuyển đổi sang dạng date

```
<p>
  | {{ birthday | date }}
</p>
```

2. Kết quả hiển thị ra trình duyệt sẽ được dữ liệu thân thiện hơn

Aug 15, 2015

3. Có thể truyền tham số cho Pipe

```
<p>
  | {{ birthday | date : "shortTime" }}
</p>
```

4. Kết quả hiển thị ra trình duyệt

12:00 AM

- Truy cập vào trang **Angular.io** để xem các pipe khác do Angular tạo sẵn
 1. Trong file ts, tạo biến json như sau:


```
person = { name: "Khoa Pham", age: 30 };
```
 2. Trong file html, hiển thị biến này ra, sử dụng pipe


```
<p>{{ person | json }}</p>
```
 3. Kết quả hiển thị ra trình duyệt


```
{ "name": "Khoa Pham", "age": 30 }
```
 4. Có thể sử dụng kết hợp nhiều Pipe liên tiếp nhau


```
<p>{{ person | json | uppercase }}</p>
```
 5. Kết quả hiển thị ra trình duyệt


```
{ "NAME": "KHOA PHAM", "AGE": 30 }
```
 6. Tạo biến kiểu Promise (là kiểu từ server trả về)


```
address = Promise.resolve("123 Pham Nhu Xuong");
```
 7. Dùng Pipe để chuyển đổi từ kiểu Promise về kiểu bình thường để có thể xem được


```
<p>{{ address | async }}</p>
```
 8. Kết quả hiển thị ra trình duyệt


```
123 Pham Nhu Xuong
```

37. Custom pipe

- Định nghĩa ra các **pipe** và sử dụng
- Ví dụ:
 1. Tạo pipe mới tên là: **round.pipe.ts** để làm tròn số, nhớ import vào **app.module.ts**
 2. Khai báo pipe này:

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

@Pipe({name: "roundNum"})

export class RoundPipe implements PipeTransform{
  transform(value: number): number{
    return Math.round(value);
  }
}
```

3. Sử dụng pipe này trong file **app.component.html**

```
<p>{{ 1.9 | roundNum }}</p>
```
 4. Kết quả hiển thị ra trình duyệt là số 1.9 được làm tròn thành 2
- Truyền tham số cho các pipe
1. Khai báo để truyền thêm tham số cho pipe

```
transform(value: number, isUpper: boolean): number{
  if(isUpper){
    return Math.ceil(value);
  }
  return Math.floor(value);
}
```

2. Sử dụng pipe và truyền thêm tham số vào

```
<p>{{ 1.9 | roundNum : true }}</p>
<p>{{ 3.5 | roundNum : false }}</p>
```

3. Truyền thêm nhiều tham số:

```
transform(value: number, isUpper: boolean, addTo: number): number {
  if (isUpper) {
    return Math.ceil(value + addTo);
  }
  return Math.floor(value + addTo);
}
```

4. Sử dụng Pipe

```
<p>{{ 1.9 | roundNum : true : 10 }}</p>
<p>{{ 3.5 | roundNum : false : 1 }}</p>
```

5. Kết quả hiển thị ra trình duyệt là số 1.9 cộng với 10 rồi sau đó làm tròn lên được 12, số 3.5 cộng với 1 rồi sau đó làm tròn xuống được 4

38. Khái niệm service và http service

- Service dùng để lấy dữ liệu từ server thông qua một API nào đó
- Thêm, xóa, sửa dữ liệu thông qua API
- Login service
- Có thể viết service chung trong component, nhưng việc tách các service ra làm cho chương trình dễ quản lý và tập trung vào các công việc cụ thể, cũng giúp dễ dàng hơn khi test các service
- Ví dụ: **Tạo một Component để lấy dữ liệu từ server, viết chung service trong component luôn**

1. Tạo component mới, đặt tên là **ip.component.ts**, nhớ import component này vào **app.module.ts**

2. Để lấy được dữ liệu từ server thì phải import thêm phương thức **Http**

```
import { Http } from "@angular/http";
```

như sau Cũng nhớ import **Http** vào **app.module.ts** luôn

```
import { HttpClientModule } from "@angular/http";
```

Thêm vào mảng import:

```
imports: [
  BrowserModule,
  AppRoutingModule,
  FormsModule,
  HttpClientModule
],
```

3. Trong file **ip.component.ts**, khởi tạo constructor để get dữ liệu từ server như sau:

```
constructor(private http: Http){
  this.http.get("http://localhost:3000/")
    .toPromise()
    .then(res => res.json())
    .then(resJson => console.log(resJson.ip))
    .catch(err => console.log(err));
}
```

4. Kết quả là trình duyệt sẽ log ra kết quả ip trả về từ đường dẫn <http://localhost:3000/>
5. Để lấy được giá trị từ server thì khai báo biến **ip** rồi gán như sau:
`.then(resJson => this.ip = resJson.ip)`
6. Sau đó dùng string interpolation để hiển thị giá trị **ip** để xem

+ Chú ý:

1. phương thức **http.get()** trả về dữ liệu dạng **Observable**, nên phải dùng **toPromise()** để chuyển về dạng Promise thì mới có thể xử lý được
2. Lệnh **then** thứ nhất trả về dữ liệu kiểu json, dùng phương thức **.json** để xử lý nó
3. Lệnh **then** thứ hai để log kết quả ra
4. Lệnh **catch** để nếu xảy ra lỗi thì log lỗi ra

39. Tách service

- Việc viết service chung trong Component làm cho code khá dài, và gây khó khăn cho việc test
- Cần phải tạo riêng một service để thực hiện chức năng get dữ liệu từ server
- Ví dụ:

1. Tạo file **ip.service.ts** để lấy dữ liệu từ server
2. Khai báo service này, có phương thức **getIp** để trả về ip lấy được:

```
import { Injectable } from "@angular/core";
import { Http } from "@angular/http";

@Injectable()

export class IpService{
  constructor(private http: Http){}

  getIp(){
    return this.http.get("http://localhost:3000/")
      .toPromise()
      .then(res => res.json())
      .then(resJson => resJson.ip);
  }
}
```

3. Khai báo service này vào mảng **providers** của component cần lấy ip
`providers: [IpService]`
4. Sử dụng service này trong component để lấy ip

```
constructor(private ipService: IpService){
  this.ipService.getIp()
    .then(ip => this.ip = ip)
    .catch(err => console.log(err));
}
```

5. Kết quả hiển thị ra trình duyệt cũng tương tự như trên

40. Sử dụng ngOnInit

- Việc sử dụng constructor để lấy dữ liệu từ service như trên là không đúng lắm, vì vậy phải sử dụng hàm ngOnInit để lấy dữ liệu, hàm constructor chỉ nên dùng để khai báo các biến mà thôi
- Các bước thực hiện:

1. Để sử dụng được phương thức **ngOnInit**, phải import thêm thư viện **OnInit**

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

2. Trong **IpComponent**, class **IpComponent** phải implements **OnInit**

```
export class IpComponent implements OnInit {
```

3. Trong class **IpComponent**, định nghĩa hàm **ngOnInit** để lấy dữ liệu từ server

```
ngOnInit(){  
    this.ipService.getIp()  
    .then(ip => this.ip = ip)  
    .catch(err => console.log(err));  
}
```

4. Cần thêm service vào mảng providers ở file **app.module.ts** để có thể sử dụng service này ở component nào cũng được

```
import { IpService } from './ip.service';
```

```
providers: [IpService],
```

41. Giới thiệu bài tập thời tiết

- Tạo giao diện nhập vào tên một thành phố bất kỳ, hiển thị ra nhiệt độ của thành phố đó

danang is now 25.84

- Nếu nhập vào thành phố không tồn tại thì cũng thông báo cho người dùng biết
- Nhiệt độ được lấy từ trang web: <https://api.openweathermap.org/data/2.5/weather?appid=1f481fd4fafd2cf9425a51599c8abd7a&units=metric&q=hanoi>, cung cấp nhiệt độ của một thành phố nào đó

42. API open weather map

43. Weather service

- Tạo giao diện, tạo ra service để lấy nhiệt độ
- **Các bước thực hiện:**

1. Tạo component mới: **ng g c weather**
2. Trong file html, tạo giao diện cho chương trình

```
<h2>Saigon is now 34</h2>  
<input placeholder="Enter your city name"><br /><br />  
<button>Get Weather</button>
```

- Trong component **weather** vừa tạo, tạo một file mới tên **weather.service** để viết service, nhớ thêm service này vào mảng **providers** của **app.module.ts**
- Trong weather service này, khởi tạo một hàm **getTemp** để trả về nhiệt độ của một thành phố, tham số nhận vào là tên một thành phố (**cityName**)

```
getTemp(cityName: String) {
  const url = "https://api.openweathermap.org/data/2.5/weather?appid=1f481fd4fafd2cf9425a51599c8abd7a&units=metric";
  return this.http.get(url)
    .toPromise()
    .then(res => res.json())
    .then(resJson => resJson.main.temp);
}
```

44. Lấy nhiệt độ và hiển thị

- Thêm sự kiện (**click**)="**getWeather()**" cho component weather để khi kích vào thì thực hiện lấy nhiệt độ
- Thêm thuộc tính **[(ngModel)]** để binding giá trị cho biến **txtCityName**

```
<input placeholder="Enter your city name" [(ngModel)]="txtCityName"><br /><br />
<button (click)="getWeather()">Get Weather</button>
```

- Định nghĩa hàm **getWeather** để lấy nhiệt độ của một thành phố, sử dụng hàm **getTemp** của weather service đã định nghĩa, tham số truyền vào là giá trị **txtCityName**

```
getWeather() {
  this.weatherService.getTemp(this.txtCityName)
    .then(temp => {
      this.temp = temp;
    })
    .catch(err => {
      console.log(err)
    });
}
```

- ở file html, thêm điều kiện để hiển thị thông báo yêu cầu nhập hoặc nhiệt độ đã lấy được

```
<h2>{{ cityName==" "? "Enter your city name" : (cityName + " is now " + temp) }}</h2>
```

- + Chú ý: Nhớ khai báo các biến **temp**, **txtCityName**, **cityName**
- Kết quả hiển thị ra trình duyệt

Saigon is now 33

45. Tính năng loading và xử lý lỗi

- Trong lúc đợi lấy dữ liệu từ server thì hiển thị thông báo cho người dùng biết là đang tải dữ liệu
- Sau khi lấy xong thì hiển thị nhiệt độ cho người dùng biết, đồng thời xóa sạch ô input ban đầu
- Nếu xảy ra lỗi thì thông báo cho người dùng biết
- Các bước thực hiện:

1. Trong file ts, tạo thêm biến `isLoading = false;`
2. Khi bắt đầu get dữ liệu gán biến `this.isLoading = true;`
3. Sau khi lấy dữ liệu xong, gán lại biến `isLoading = false;`

Cụ thể hàm `getWeather` sẽ như sau:

```
getWeather() {  
  this.isLoading = true;  
  this.weatherService.getTemp(this.txtCityName)  
    .then(temp => {  
    this.cityName = this.txtCityName;  
    this.temp = temp;  
    this.isLoading = false;  
  })  
  .catch(err => {  
    console.log(err)  
  });  
}
```

4. Định nghĩa hàm `getMessage` để hiển thị thông báo

```
getMessage(){  
  if(this.isLoading){  
    return "Đang tải dữ liệu..."  
  }  
  return this.cityName==" "? "Enter your city name" : (this.cityName + " is now " + this.temp)  
}
```

5. Trong file html, hiển thị thông báo bằng hàm `getMessage`

```
<h2>{{ getMessage() }}</h2>
```

6. Nếu nhập vào thành phố không tồn tại, biến `isLoading` luôn có giá trị `true`, nên cứ loading mãi, để tránh điều này, cần xử lý ở phần `catch` như sau:

```
.catch(err => {  
  alert("Can't find your city name!");  
  this.isLoading = false;  
});
```

7. Đồng thời sau khi hoàn thành get dữ liệu, phải gán lại giá trị ban đầu cho các biến

```

getWeather() {
  this.isLoading = true;
  this.weatherService.getTemp(this.txtCityName)
    .then(temp => {
      this.cityName = this.txtCityName;
      this.temp = temp;
      this.isLoading = false;
      this.txtCityName = "";
    })
    .catch(err => {
      alert("Can't find your city name!");
      this.isLoading = false;
      this.txtCityName = "";
      this.cityName = "";
    });
}

```

46. Giới thiệu về form

- Hầu hết các ứng dụng đều thu thập thông tin của người dùng thông qua các form
- Angular hỗ trợ việc xử lý form input rất tốt
- Ví dụ:
 1. Tạo một component mới tên là **sign-in.component.ts**
 2. Trong **template** của component này, hiển thị 2 thẻ input để nhập **email** và **password**, và 1 button để thực hiện submit form

```

template: `
  <p>This is Sign in Form</p>
  <input placeholder="Email" [(ngModel)] = "email" />
  <br /><br />
  <input type="password" placeholder = "Password" [(ngModel)]="password" />
  <br /><br />
  <button>Submit</button>
`

```

3. Thêm sự kiện (click) cho button Submit, và định nghĩa hàm **onSubmit**, để log ra giá trị email và password nhập vào từ ô input

```

onSubmit(){
  console.log("sign in form: ", this.email, this.password);
}

```

4. Kết quả, sau khi nhập thông tin vào ô input và nhấn submit, trình duyệt sẽ log ra giá trị email và password nhập vào từ ô input

47. ngForm: tài liệu tham khảo: <https://angular.io/guide/forms>

- có 2 kiểu form thường được sử dụng là **Template-driven Form** và **Reactive Form**
- Trong bài này sẽ tìm hiểu về **Template-driven Form**
- Trong form thì có các thẻ input để nhập thông tin
- Có sự kiện (**submit**) để submit giá trị của form
- Có thuộc tính **ngModel**, và kèm theo là thuộc tính **name**

- Mỗi form đều được gán 1 giá trị là **ngForm**

- **Ví dụ:**

1. Tạo một component mới tên là **sign-in2**
2. Khai báo 1 form chứa 2 thẻ input, mỗi thẻ input có thuộc tính **[(ngModel)]** để binding giá trị cho ô input và kèm theo thuộc tính **name**
3. Gán sự kiện **(submit)="onSubmit()"** cho thẻ form, gán biến **#SignInForm="ngForm"**

```
template: `
  <p>This is Sign in Form2</p>
  <form (submit)="onSubmit(SignInForm)" #SignInForm="ngForm">
    <input placeholder="Email" ngModel name="email" />
    <br /><br />
    <input type="password" placeholder = "Password" ngModel name="password" />
    <br /><br />
    <input type="submit" value="Submit">
  </form>
`
```

4. Định nghĩa hàm **onSubmit** để log ra giá trị nhập vào từ ô input

```
onSubmit(SignInForm){
  console.log("sign in form2: ", SignInForm.value.email, SignInForm.value.password)
}
```

5. Kết quả, sau khi nhập thông tin vào ô input và nhấn submit, trình duyệt sẽ log ra giá trị email và password nhập vào từ ô input

48. **Validate template form**

- Validate là kiểm tra dữ liệu nhập vào trước khi submit lên server
- Giúp cho người dùng biết được trường nào họ nhập vào có thể sai
- **Ví dụ:** có những trường yêu cầu phải nhập, có trường nhập sai định dạng...

- **Ví dụ áp dụng:**

1. Tạo component mới tên là **ng-form.component.ts**, nhớ import vào app.module.ts
2. Tạo form có 2 thẻ input để nhập **email** và **password**
3. Thêm thuộc tính **required** cho thẻ input **email**

```
<p>This is ngForm</p>
<form (submit)="onSubmit(SignInForm)" #SignInForm="ngForm">
  <input placeholder="Email" ngModel name="email" required />
  <br />
  <input type="password" placeholder = "Password" ngModel name="password" />
  <br /><br />
  <button>Submit</button>
</form>
```

4. Kết quả hiển thị ra trình duyệt

This is ngForm

Submit

5. Khi chưa nhập thông tin cho thẻ email thì giá trị của thuộc tính invalid của thẻ này là false, do đó, giá trị thuộc tính của form cũng là form

6. Thêm thuộc tính **[disabled]** vào thẻ button để khi giá trị form invalid thì ẩn nút submit đi

```
<button [disabled]="SignInForm.invalid">Submit</button>
```

7. Thêm điều kiện để nếu trường email **có lỗi** thì hiện thông báo

```
<p *ngIf="SignInForm.controls.email?.errors?.required">
Email is invalid
</p>
```

8. Kết quả hiển thị ra trình duyệt, nếu trường email chưa nhập thì hiện thông báo lỗi, và nếu form invalid thì ẩn nút submit đi

This is ngForm





Email is invalid

Submit

*chú ý: Phải thêm điều kiện `email?.errors?.required` để nếu email là **null** hoặc **undefined** thì không truy cập tiếp (khỏi xảy ra lỗi)

49. ngSubmit

- các thuộc tính của ngForm thường đi theo cặp, và có giá trị đối nhau

- a. **dirty: true**  **pristine: false** ô input đã nhập chưa?
- b. **touched: true**  **untouched: false** đã chạm vào ô này chưa?
- c. **valid: true**  **invalid: false** đã valid ô này chưa?
- d. **submitted: true**  **pending: false** đã submit chưa?

- Khi thực hiện `throw new Error("Form is invalid!");` để ném ngoại lệ, thì giá trị của form sẽ được submit đi
- **Sử dụng** (ngSubmit) để khi thêm câu lệnh throw ở file ts thì form chỉ ném ra ngoại lệ ở console mà không cho submit, tức là không tải lại trang

50. Xử lý style các input không hợp lệ

1. Tạo biến template variable để giúp câu lệnh truy cập đến các thuộc tính ngắn gọn hơn
2. Thêm validate cho thẻ input là phải đúng định dạng email
3. Nếu ô input này invalid và đã touched vào thì hiển thị câu thông báo lỗi

```
<input placeholder="Email" ngModel #txtEmail="ngModel" name="email" required email
<p *ngIf="txtEmail.errors?.required && txtEmail.touched">
Email is required
</p>
<p *ngIf="txtEmail.errors?.email && txtEmail.touched">
Email is invalid
</p>
```

4. Cuối cùng hiện thông báo lỗi ở dưới cho người dùng biết

```
<p>{{ txtEmail.errors | json }}</p>
```

5. Kết quả hiển thị ra trình duyệt, ban đầu thì chỉ hiện thông báo lỗi ở dưới là required: true, Nếu touched vào ô email thì hiển thị thông báo lỗi ở dưới ô này, nếu nhập email sai định dạng cũng hiện thông báo lỗi

<p>This is ngForm</p> <div>Email</div> <div>Password</div> <div>Submit</div> <p>{ "required": true }</p>	<p>This is ngForm</p> <div>nvding185</div> <p>Email is invalid</p> <div>Password</div> <div>Submit</div> <p>{ "email": true }</p>
--	---

- Xử lý style các input không hợp lệ

1. Vào inspect để biết class của những ô không hợp lệ

```
<input email name="email" ngmodel placeholder="Email" required
ng-reflect-required ng-reflect-email ng-reflect-name="email" ng-
reflect-model class="ng-invalid ng-touched ng-dirty"> == $0
```

2. Dựa vào class để định dạng CSS cho các thẻ này trong file **style.css**

```
input.ng-invalid.ng-touched{
border: 1px solid red;
}
```

3. Kết quả hiển thị ra trình duyệt, khi ô input invalid và đã touched vào thì sẽ bị tô màu viền đỏ

51. Xử lý checkbox và ngModelGroup

a. checkbox

- Thêm thẻ **checkbox** vào cho form

```
<label><input type="checkbox" ngModel name="football">Football</label><br />
```

- Hiện thị giá trị form ngay dưới thẻ form luôn

```
<p>{{ SignInForm.value | json }}</p>
```

- Kết quả hiển thị: ngay khi mở trình duyệt lên thì giá trị đều là null

This is ngForm

Email
Password

☐ Football

```
{ "required": true }
```

```
{ "email": "", "password": "", "football": "" }
```

- Dùng event-binding để gán giá trị ban đầu là false cho thẻ checkbox

```
<label><input type="checkbox" [ngModel]="false" name="football">Football</label><br />
```

b. ngModelGroup

- Tạo 1 thẻ div chứa 3 thẻ checkbox là tên 3 môn học
- Thêm thuộc tính **ngModelGroup** cho thẻ div
- Khi đó hình thành một đối tượng mới chứa các môn học

```
<div ngModelGroup="subjects">
  <label><input type="checkbox" [ngModel]="false" name="NodeJS">NodeJS</label>
  <label><input type="checkbox" [ngModel]="false" name="Angular">Angular</label>
  <label><input type="checkbox" [ngModel]="false" name="ReactJS">ReactJS</label>
</div>
```

- Kết quả hiển thị ra trình duyệt:

This is ngForm

Email
Password

☐ NodeJS ☐ Angular ☐ ReactJS

```
{ "required": true }
```

```
{ "email": "", "password": "", "subjects": { "NodeJS": false, "Angular":
false, "ReactJS": false } }
```

c. Các loại validator

- Vào đường dẫn <https://angular.io/api?query=validator> để xem các loại validator của Angular
- Thêm validator **minlength** cho input **password**

```
<input
  type="password"
  placeholder = "Password"
  ngModel
  #txtPassword = "ngModel"
  name="password"
  minlength="6"
/>
```

- Hiện lỗi ngay dưới form để xem

```
<p>{{ txtPassword.errors | json }}</p>
```

- Kết quả: Nếu nhập vào password bé hơn 6 ký tự thì sẽ hiển thị thông báo lỗi

```
{ "minlength": { "requiredLength": 6, "actualLength": 2 } }
```
 - Thêm validator **pattern** = "[a-z]*" cho ô input **password**, có nghĩa là ô này chỉ được nhập các chữ cái từ a đến z `pattern = "[a-z]*"`
 - Kết quả: nếu nhập vào password không thỏa mãn thì sẽ hiển thị 2 lỗi

```
{ "minlength": { "requiredLength": 6, "actualLength": 3 }, "pattern": { "requiredPattern": "^[a-z]*$", "actualValue": "ff8" } }
```
- Nếu nhập vào password thỏa mãn thì lỗi là **null**

52. Reactive form

a. Reactive Forms là gì?

- **Reactive** Forms là một phương pháp để tạo form trong Angular
- Trong **Reactive** forms, chúng ta tạo toàn bộ form control ở trong code (khởi tạo ngay, khởi tạo trong constructor, hoặc khởi tạo trong ngOnInit), nên có thể dễ dàng truy cập các phần tử của form ngay tức thì.

b. Các thành phần cơ bản của form

- AbstractControl là một abstract class cho 3 lớp con: **FormControl**, **FormGroup**, và **FormArray**.
- 1. **FormControl** là đơn vị nhỏ nhất, nó lưu giữ giá trị và trạng thái hợp lệ của một form control. Tương ứng với một HTML form control như **input**, **select**.
- 2. **FormGroup** nó lưu giữ giá trị và trạng thái hợp lệ của một nhóm các đối tượng thuộc AbstractControl – có thể là **FormControl**, **FormGroup**, hay **FormArray**
- 3. **FormArray** nó lưu giữ giá trị và trạng thái hợp lệ của một mảng các đối tượng thuộc AbstractControl giống như **FormGroup**.

c. Tạo file templates:

```

<? signup.component.html x TS signup.component.ts
1 <p>This is SignUp Form</p>
2 <form (submit)="onSubmit()" [formGroup]="formSignUp">
3   <input placeholder="Email" formControlName="email" />
4   <br /><br />
5   <input type="password" placeholder="Password" formControlName="password" />
6   <br />
7   <div formGroupName="subject">
8     <label> NodeJS<input type="checkbox" formControlName="nodeJS" /></label>
9     <label> Angular<input type="checkbox" formControlName="angular" /></label>
10    <label> ReactJS<input type="checkbox" formControlName="react" /></label>
11  </div>
12  <br />
13  <button>Submit</button>
14 </form>

```

d. Import APIs cho Reactive forms

- Để có thể sử dụng các APIs mà Angular cung cấp cho việc thao tác với Reactive forms, chúng ta cần import NgModule là **ReactiveFormsModule** từ package **@angular/forms** như sau:

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

e. Khởi tạo form trong Component

- khởi tạo trong ngOnInit.

```

ngOnInit() {
  this.formSignUp = new FormGroup({
    email: new FormControl('dinh@gmail.com'),
    password: new FormControl(),
    subject: new FormGroup({
      nodeJS: new FormControl(),
      angular: new FormControl(true),
      react: new FormControl(false)
    })
  })
}

```

f. Reactive Forms Validator

- Việc thêm Validator vào cho một control rất đơn giản, việc của bạn là thêm giá trị tham số tiếp theo như sau:

```
email: ['', [Validators.required, Validators.minLength(3)]],
```

Ở đoạn code trên, chúng ta gộp **required** và **minLength(3)** cho control **email**, hoặc có thể thêm nhiều các validator function nữa theo ý muốn.

- Để hiển thị thông báo cho người dùng biết về lỗi, chúng ta thêm đoạn code sau trong file html

```

<p *ngIf="formSignUp.get('email').invalid && formSignUp.get('email').touched">
  Email is required
</p>

```

g. Form Builder

- **Form Builder** giúp tạo form model một cách nhanh chóng và tiện lợi
- **Import** FormBuilder vào class muốn tạo form

```
import { FormGroup, FormBuilder } from '@angular/forms'
```

- gọi các API như **group**, **array**, **control** để tạo form.

```
formSignUp: FormGroup;
constructor(private fb: FormBuilder) {}

ngOnInit() {
  this.formSignUp = this.fb.group({
    email: '',
    password: '',
    subject: this.fb.group({
      nodeJS: true,
      angular: false,
      react: false
    })
  })
}
```

h. Tùy chỉnh ngoại lệ Reactive form

- Là tạo ra các **validator** do người lập trình quy định
- Tạo ra các Validator cũng như tạo các function bình thường khác
- Ví dụ dưới đây tạo một Validator cho một FormControl, để quy định email nhập vào phải đúng định dạng gmail

```
function gmailValidator(formControl: FormControl) {
  if(formControl.value.includes('@gmail.com')) return null;
  else return { gmail: true };
}
```

- Việc thêm Validator vào cho một control tương tự như thêm các Validator mặc định khác

```
email: ['', [Validators.required, gmailValidator]],
```

- Muốn hiển thị xem Validator có lỗi gì không, ta thêm đoạn code này vào file html

```
<code>{{ formSignUp.controls.email.errors | json }}</code>
```

- Muốn sử dụng nhiều Validator cho một biến nào đó, thêm vào mảng Validator như sau:

```
email: ['', [Validators.required, gmailValidator]],
```