

TRƯỜNG ĐẠI HỌC SƯ PHẠM KỸ THUẬT HƯNG YÊN
KHOA CÔNG NGHỆ THÔNG TIN



ĐỀ CƯƠNG HỌC PHẦN
CÔNG NGHỆ WEB VÀ ỨNG DỤNG

HƯNG YÊN NĂM 2020

MỤC LỤC

Bài 1: Tổng quan về lập trình front end với Angular	5
1.1 Giới thiệu về Angular	5
1.2 Setup môi trường phát triển ứng dụng.....	10
1.3 Tạo project đầu tiên	13
1.4 Các thành phần trong một dự án Angular.....	16
1.5 Kiến trúc một ứng dụng Angular.....	18
1.6. Một số câu lệnh cơ bản của Angular Cli	27
Bài 2: Cơ bản về ngôn ngữ lập trình TypeScript.....	28
2.1 TypeScript là gì?.....	28
2.2 Tại sao nên sử dụng Typescript	28
2.3 Các kiểu dữ liệu cơ bản và cách khai báo biến.....	30
2.3.1 Các kiểu dữ liệu cơ bản	30
2.3.2 Cách khai báo biến	32
2.4 Các cấu trúc điều khiển	34
2.4.1 Lệnh If Else trong TypeScript	34
2.4.2 Lệnh Switch Case trong TypeScript.....	36
2.4.3 Vòng lặp trong TypeScript	38
2.5 Hàm và cách khai báo.....	40
2.5.1 Khai báo.....	40
2.5.2 Function type (kiểu hàm):.....	41
2.5.3 Inferring the type (suy kiểu)	41
2.5.4 Optional and default parameter (tham số tùy chọn và mặc định).....	41
2.4.5 Rest Parameter	42
2.5.6 Arrow function	43
2.6 Khám phá chi tiết một số kiểu dữ liệu.....	44
2.6.1 Kiểu dữ liệu string	44
2.6.2 Kiểu dữ liệu Arrays	46
2.7 Lập trình hướng đối tượng.....	52
Bài 3: Component.....	59
3.1 Component là gì?.....	59
3.2 Thành phần cha AppComponent	60
3.3 Tạo một Component	61
3.4 Cấu trúc chung của một Component	62
3.5 Sử dụng component	63
3.6 Nhúng bootstrap, css, js vào project.....	67
3.7 Module.....	68
Bài 4: Template và Data Binding	71

4.1 String interpolation	71
4.2 Property Binding	71
4.3 Event Binding	72
4.4 Two-way binding.....	75
4.5 Vòng đời component	76
4.6 Giao tiếp giữa các component bằng @Input và @Output.....	77
4.7 Làm việc với @ViewChild.....	80
Bài 5: Template và Data Binding(tiếp).....	83
5.1 Directive là gì?.....	83
5.2 Structure Directive: ngIf, ngFor, ngSwitch, Ng-Template, Ng-Container.....	83
5.3 Attribute Directive: ngStyle, ngClass, ngContainer	86
5.4 Pipes	89
5.5 Animation	93
Bài 6: Routing & Navigation.....	98
6.1 Cơ bản về routing	98
6.2 Lazy Loading	111
Bài 7: Thảo luận thiết kế và cài đặt website giới thiệu và đặt hàng online(buổi 1)	122
Bài 8: Xử lý Form.....	123
8.1 Template Driven Form	123
8.1.1 Template	123
8.1.2 Import APIs cho Template-driven forms.....	125
8.1.3 ngForm và ngModel directives.....	125
8.1.4 ngModelGroup directive.....	128
8.1.5 Submit form.....	130
8.1.6 Template-driven error validation.....	130
8.2 Model Driven Form(Reactive Form).....	133
8.2.1 Các thành phần cơ bản của form	133
8.2.2 Template	134
8.2.3 Import APIs cho Reactive forms	136
8.2.4 Khởi tạo form trong Component	137
8.2.5 Binding controls	138
8.2.6 Reactive Forms Validator	139
8.2.7 Form Builder	142
8.2.8 FormArray	143
8.2.9 Forms with a single control	145
8.2.10 Cập nhật giá trị cho form, control	145
Bài 9: Services & Dependency Injection.....	147
9.1 Service là gì?	147

9.2 Dependency Injection (DI)	147
9.3 Viết service dưới dạng dùng chung	148
9.4 HttpClient	149
9.4.1 Setup: installing the module	150
9.4.2 Making a request for JSON data	150
9.4.3 Error handling	151
9.4.4 Getting error details	152
9.4.5 Requesting non-JSON data.....	153
9.4.6 Sending data to the server Making a POST request:	153
9.4.7 Configuring other parts of the request:	153
9.4.8 URL Parameters	154
9.4.9 Advanced usage.....	154
Bài 10: Thảo luận thiết kế và cài đặt website giới thiệu và đặt hàng online(buổi 2)	156
Bài 11: Thảo luận thiết kế và cài đặt website giới thiệu và đặt hàng online(buổi 3)	157
Bài 12: Phân quyền người dùng	158
12.1 JWT là gì?.....	158
12.2 Cách thức phân quyền người dùng trong Angular	158
Bài 13: Thảo luận thiết kế và cài đặt website giới thiệu và đặt hàng online(buổi 4)	168
Bài 14: Thảo luận thiết kế và cài đặt website giới thiệu và đặt hàng online(buổi 5)	168
Bài 15: Tổng kết bài học	168

Bài 1: Tổng quan về lập trình front end với Angular

1.1 Giới thiệu về Angular

Nếu đã từng xây dựng một vài ứng dụng Web, chắc hẳn bạn đã từng nghe đến cái tên Angular, một Frameworks Javascript giúp chúng ta xây dựng ứng dụng Web đầy đủ tính năng từ phía Client. Angular lần đầu được phát hành bởi gã khổng lồ Google vào năm 2010 với phiên bản AngularJS, sau đó đã có chỗ đứng khá vững chắc trong một thời gian dài, phiên bản Angular 2 phát hành năm 2016 mang đến một bước chuyển mình vượt bậc, một công cụ thực sự mạnh mẽ cho việc phát triển ứng dụng Web trên cả nền tảng Mobile và Desktop. Hiện nay chúng ta đang có phiên bản Angular 10+. Do vậy toàn bộ code angular được đề cập trong cuốn đề cương này được cài đặt theo Angular 10.

Angular là một Javascript Framework dùng để xây dựng các Single Page Application (SPA) bằng JavaScript, HTML và TypeScript. Angular cung cấp các tính năng tích hợp cho animation, http service, materials và có các tính năng như auto-complete, navigation, toolbar, menus, ... Code được viết bằng TypeScript, biên dịch thành JavaScript và hiển thị giống nhau trong trình duyệt.

Hiện giữa vô vàn các thư viện và frameworks Javascript, Angular có gì nổi bật và tại sao chúng ta nên chọn Angular cho ứng dụng tiếp theo của mình? Với những lý do sau đây:

*** Angular giúp nâng cao năng suất của các lập trình viên.**

Việc phát triển Web đã có bước thay đổi đáng kể trong vài năm qua. Với phiên bản ECMAScript (ES) 2015 - chúng ta quen thuộc với cái tên ES6, với những class hay arrow function. Angular 2+ ứng dụng những tính năng mới này giúp việc code với Angular trở nên rõ ràng và dễ học hơn rất nhiều. Thêm vào đó, với việc ứng dụng Typescript - một ngôn ngữ - hay là một bản nâng cấp đáng giá của Javascript, Angular kết hợp với Typescript, chúng ta có một công cụ tuyệt vời giúp xử lý các vấn đề hạn chế của JS như kiểm tra kiểu dữ liệu, refactor code an toàn hơn, ... từ đó cũng hỗ trợ tốt hơn cho việc Debug cũng như giúp các Dev thực sự hiểu rõ mã nguồn của họ hơn.

*** Cấu trúc phát triển rõ ràng.**

Điều quan trọng của một Frameworks đối với lập trình viên đó là cấu trúc phát triển ứng dụng của nó, và Angular mang đến một kiến trúc rất rõ ràng, dựa trên ba yếu

tổ chính: class, các dependency được thêm vào và mô hình MVVM (model-view-view/model). Angular sử dụng class trong ES6 với một loạt các thuộc tính để xây dựng toàn bộ các cấu trúc chủ chốt, giả sử bạn muốn tạo một Angular component - Tạo một class và thêm vào các thuộc tính cần thiết. Hay bạn muốn tạo một Angular module - Hãy tạo một class và thêm vào đó các thuộc tính cần thiết. Về cơ bản sẽ là như vậy, Angular cung cấp một cấu trúc rõ ràng để xây dựng từng tính năng cho ứng dụng của bạn. Các dependency mạnh mẽ được sử dụng trong ứng dụng khi cần thiết, và khi cần tích hợp bất kì dependency nào, như HTTP hay Router, chúng ta chỉ cần thêm nó vào bên trong constructor của class. Mô hình MVVM cũng giúp Angular chiếm lợi thế trong xây dựng ứng dụng client-side, thường ta sẽ có 3 điều cần quan tâm chính: đó là giao diện người dùng, mã nguồn điều khiển giao diện và mô hình dữ liệu (data) cho giao diện. Angular với MVVM phân biệt hoàn toàn rõ ràng các yếu tố trên nhờ mô hình MVVM:

Phần giao diện (view) được định nghĩa trong một template bao hàm HTML dành cho một component nhất định. Template có thể là toàn bộ Layout hoặc bất cứ mảnh ghép nào trong Layout đó.

Model được định nghĩa như là các thuộc tính của component class. Có thể hiểu là dữ liệu, dựa vào đó để phần View sử dụng để thực thi.

View/model là class quản lý cả view cũng như model. Là phần code sẽ xử lý việc truy xuất dữ liệu, đồng thời thực thi các tương tác của người dùng trên view.

Với việc ứng dụng các điểm tích cực của các thành phần trên, Angular khiến việc phát triển ứng dụng trở nên dễ dàng và hiệu quả hơn.

*** Extensive binding**

Rất nhiều ứng dụng Web làm việc với dữ liệu (data). App sẽ truy xuất dữ liệu từ Server và hiển thị dữ liệu đó tới người dùng trên view, sử dụng template. Và các tương tác của người dùng sẽ được khiến dữ liệu thay đổi, được view ghi nhận và lưu lại trên server. Data Binding trong Angular giúp bạn thực thi tiến trình trên rất dễ dàng. Đơn thuần từ việc ràng buộc thành phần HTML trong template với các thuộc tính trong class và dữ liệu sẽ tự động xuất hiện trên màn hình. Với các tương tác của người dùng đòi hỏi thay đổi dữ liệu, Angular sử dụng phương pháp two-way binding. Bất kì thay đổi dữ liệu đến từ view sẽ tự động cập nhật thuộc tính "model" bên trong class. Thêm vào đó, Angular cũng hỗ trợ property binding - cho phép chúng ta điều khiển DOM

bằng cách ràng buộc thuộc tính HTML với thuộc tính của component class, data sẽ tự động xuất hiện bên trong view. Ví dụ, chúng ta ràng buộc thuộc tính hidden đối với một thẻ img với thuộc tính hideImg bên trong class. Khi thuộc tính hideImg nhận giá trị true, img sẽ tự động hidden và ngược lại khi hideImg nhận giá trị false, thẻ img sẽ tự động hiển thị tới người dùng.

Cuối cùng, Angular hỗ trợ event binding, có nghĩa là chúng ta có thể xử lý bất kì event nào từ phía view, như HTML event. Về cơ bản chúng ta sẽ gán event với một method bên trong class. Mỗi khi event xuất hiện, method tương ứng sẽ được thực thi.

Extensive binding giúp quá trình hiển thị dữ liệu, điều khiển DOM, thực thi các event một cách trơn tru và dễ dàng.

*** Hỗ trợ đầy đủ tính năng điều hướng (routing)**

Đa số các ứng dụng Web không chỉ có 1 view hay một page duy nhất, mà sẽ cung cấp nhiều view khác nhau tương ứng với các chức năng chính. Ví dụ như một trang web với các trang giới thiệu, trang nội dung, trang chi tiết, trang đăng nhập, đăng ký,... Chúng ta sẽ cần hiển thị đúng view vào đúng thời điểm. Đó là mục đích của điều hướng (routing). Và Angular cung cấp đầy đủ tính năng cho việc này, chúng ta định nghĩa các đường dẫn (route) cho mỗi page view của ứng dụng. Và chúng ta sẽ kích hoạt route dựa trên tương tác của người dùng (user). Chúng ta có thể truyền thêm dữ liệu vào các route, giúp view hiển thị nội dung một cách dynamic, có thể bảo vệ route để người dùng chỉ có thể truy cập sau khi đã đăng nhập hoặc có quyền truy cập, có thể ngăn chặn việc người dùng ngay lập tức rời một trang khi các thao tác còn dang dở cho đến khi họ thực sự xác nhận việc rời đi hoặc lưu lại tiến trình sử dụng,...Angular đồng thời cũng hỗ trợ child-route cho việc điều hướng bên trong một route. Việc điều hướng giữa các view bên trong ứng dụng Angular thực sự rất linh hoạt và mạnh mẽ.

*** Angular giúp giảm tối đa kích thước và tăng tối đa hiệu suất của ứng dụng.**

Kích thước và hiệu năng có mối liên quan mật thiết khi chúng ta làm việc trên nền tảng Web. Một component nhỏ hơn sẽ giúp nâng cao hiệu suất khởi động - giảm cả thời gian download cũng như thời gian cũng như thời gian compile trên trình duyệt. Giảm kích thước component và giúp tăng hiệu suất là một ưu điểm cũng như mục tiêu mà Angular mong muốn mang đến cho các lập trình viên.

Giảm kích thước ứng dụng có thể thực hiện bằng nhiều cách. Đầu tiên chúng ta có thể giảm tối đa kích thước của từng component tới mức tối thiểu có thể. Tiếp theo các

component sẽ được sắp xếp bên trong Angular Module bằng 1 cách để cho các nhóm logic có liên quan đến nhau sẽ được download cùng với nhau. Và bước thứ ba, lazy loading bên trong các route sẽ chỉ download những module cần thiết cho việc hiển thị nội dung cần thiết tới người dùng, và sẽ không bao giờ download những nội dung không cần thiết.

Chúng ta có một trình biên dịch tên là AOT, trình biên dịch này sẽ chạy một lần trong thời gian build ứng dụng. Trình duyệt sau đó sẽ download phiên bản chưa được biên dịch của ứng dụng và render ứng dụng tới người dùng ngay lập tức mà không cần biên dịch nó lần đầu trong trình duyệt. Thêm nữa là sẽ không cần download trình biên dịch Angular, giúp làm giảm đáng kể kích thước (size) của ứng dụng cần tải về.

*** Document và cộng đồng (community)**

Document cho Angular 2+ - angular.io - rất đầy đủ và chi tiết, bao hàm giới thiệu cơ bản giúp bạn làm quen nhanh chóng với Angular, giới thiệu chi tiết, từ cơ bản đến nâng cao các API của Angular, cũng như có hẳn một Tutorial Basic được xây dựng nên bởi Angular team, cung cấp cho bạn nhanh chóng nắm bắt các thuộc tính cơ bản của Framework.

Angular cũng có một cộng đồng sử dụng rất lớn, đồng thời được phát triển bởi gã khổng lồ Google, khiến Angular không ngừng trưởng thành với các phiên bản luôn được cập nhật, hiện chúng ta đã có phiên bản Angular 10+ mới cam kết LTS từ Google (*Các phiên bản có gắn mác LTS sẽ là phiên bản được hỗ trợ dài hạn. Tức là cho dù đã phát hành được một thời gian và kể cả đã có phiên bản khác mới hơn được phát hành thì phiên bản có gắn mác LTS sẽ vẫn được hỗ trợ bảo trì như bình thường*)

*** Giới thiệu các version angular**

AngularJs

Phiên bản đầu tiên của angular là AngularJs được bắt đầu từ năm 2009 và đc ra mắt vào 20/10/2010 bởi lập trình viên Misko Hevery tại Google. Lúc đó AngularJs được viết theo mô hình MVC (Model-View-Controller) trong đó:

- + Model là thành phần trung tâm thể hiện hành vi của ứng dụng và quản lý dữ liệu.
- + View được tạo ra dựa trên thông tin của Model.
- + Controller đóng vai trò trung gian giữa Model và View và để xử lý logic.

Angular 2

Ra mắt tháng 3 năm 2015 phiên bản Angular 2 ra đời nhằm thay thế AngularJs với các khái niệm mới nhằm đơn giản hóa và tối ưu cho quá trình phát triển sử dụng framework này. Angular 2 thay đổi hoàn toàn so với AngularJs bằng việc thay Controllers và \$scope (AngularJs) bằng Components và Directives. Components = Directives + Template, tạo nên view của ứng dụng và xử lý các logic trên view. Angular 2 hoàn toàn được viết bằng Typescript. Angular 2 nhanh hơn AngularJs, hỗ trợ đa nền tảng đa trình duyệt, cấu trúc code được tổ chức đơn giản và dễ sử dụng hơn.

Angular 4

Ra mắt vào tháng 3/2017 đây là một phiên bản nâng cấp từ Angular 2 nên kiến trúc không thay đổi nhiều ngoài việc giảm thiểu code được tạo ra từ đó giảm kích thước tệp được đóng gói xuống 60%, đẩy nhanh quá trình phát triển ứng dụng.

Angular 5

Đã được phát hành vào ngày 1 tháng 11 năm 2017 với mục tiêu thay đổi về tốc độ và kích thước nên nó nhanh hơn và nhỏ hơn angular 4. Các tính năng mới so với angular 4:

- + Sử dụng HTTPClient thay vì sử dụng HTTP : bởi vì nó nhanh, an toàn và hiệu quả hơn.
- + Với phiên bản Angular 5 mặc định sử dụng RxJs 5.5
- + Multiple export aliases : Một component có thể được xuất bằng nhiều bí danh (aliases) để giảm bớt quá trình di chuyển.
- + Internationalized Pipes for Number, Date, and Currency: Các pipe mới được giới thiệu để tiêu chuẩn hóa tốt hơn.
- + Tối ưu hóa build production bằng việc sử dụng công cụ build optimizer được tích hợp sẵn vào trong CLI. Công cụ này tối ưu tree shake và loại bỏ code dư thừa.
- + Cải thiện tốc độ biên dịch bằng việc dùng TypeScript transforms, giờ đây khi build sẽ sử dụng lệnh “ng serve –aot”. AOT sẽ cải thiện performance khi load page và nó được dùng để deploy app lên production

Angular 6

Được phát hành vào ngày 4 tháng 5 năm 2018 với những thay đổi so với Angular 5:

- + Cập nhật CLI, command line interface: thêm 1 số lệnh mới như ng-update để chuyển từ version trước sang version hiện tại; ng-add để thêm các tính năng của ứng dụng để trở thành một ứng dụng web tiến bộ.
- + Angular Element: Cho phép các component của Angular được triển khai dưới dạng component web, sau đó có thể được sử dụng trong bất kỳ trang HTML nào một cách dễ dàng.
- + Multiple Validators: cho phép nhiều Validators được áp dụng trên form builder.
- + Tree-shakeable providers: giúp loại bỏ mã code chết.
- + Sử dụng RxJS 6 với syntax thay đổi.

Angular 7

Được phát hành vào 18 tháng 10 năm 2018 với những thay đổi như:

- + Support TypeScript 3.1
- + ScrollingModule: Để scroll load dữ liệu.
- + Drag and Drop: Chúng ta có thể dễ dàng thêm tính năng kéo và thả vào một mục
- + Angular 7.0 đã cập nhật RxJS 6.3

Angular 8

Angular 8 sẽ phát hành với các tính năng của IVY engine. Angular 8 là phiên bản quan trọng từ Google trong năm 2019 nó tập trung vào các công cụ giúp Angular dễ dàng sử dụng cho các nhà phát triển xây dựng các loại ứng dụng với sự tối ưu về hiệu năng. Ngoài ra, phiên bản này cũng chứa một số tính năng mới được kỳ vọng từ các phiên bản trước. Angular 8 sẽ hỗ trợ phiên bản TypeScript 3.4 giúp code dễ và nhanh hơn rất nhiều.

Angular 9, Angular 10 ...

1.2 Setup môi trường phát triển ứng dụng

Angular là một nền tảng cung cấp một cách thức để dễ dàng xây dựng ứng dụng chạy trên web, mobile và cả desktop. Angular là một framework sử dụng ngôn ngữ lập trình TypeScript tạo ra ứng dụng, khi chạy nó được biên dịch sang Javascript.

Để xây dựng được một ứng dụng Angular chúng ta cần cài đặt NodeJS, NPM, Angular CLI, IDE(VS Code, WebStorm, Atom, Netbean,...)

a) Cài đặt NodeJS

NodeJS cung cấp lệnh cho phép biên dịch TypeScript sang Javascript.

Đầu tiên, bạn cần kiểm tra xem nodejs đã cài đặt trong máy tính chưa. Bạn mở terminal (đối với windows: biểu tượng windows + r), rồi gõ lệnh:

```
node -v
```




Nếu NodeJS đã tồn tại sẽ hiện thị thông tin version thì chúng ta chuyển sang bước tiếp theo

Nếu chưa tồn tại chúng ta chúng ta vào trang chủ của Node và tải trình Installer về cài đặt tại địa chỉ <https://nodejs.org/en/download/>

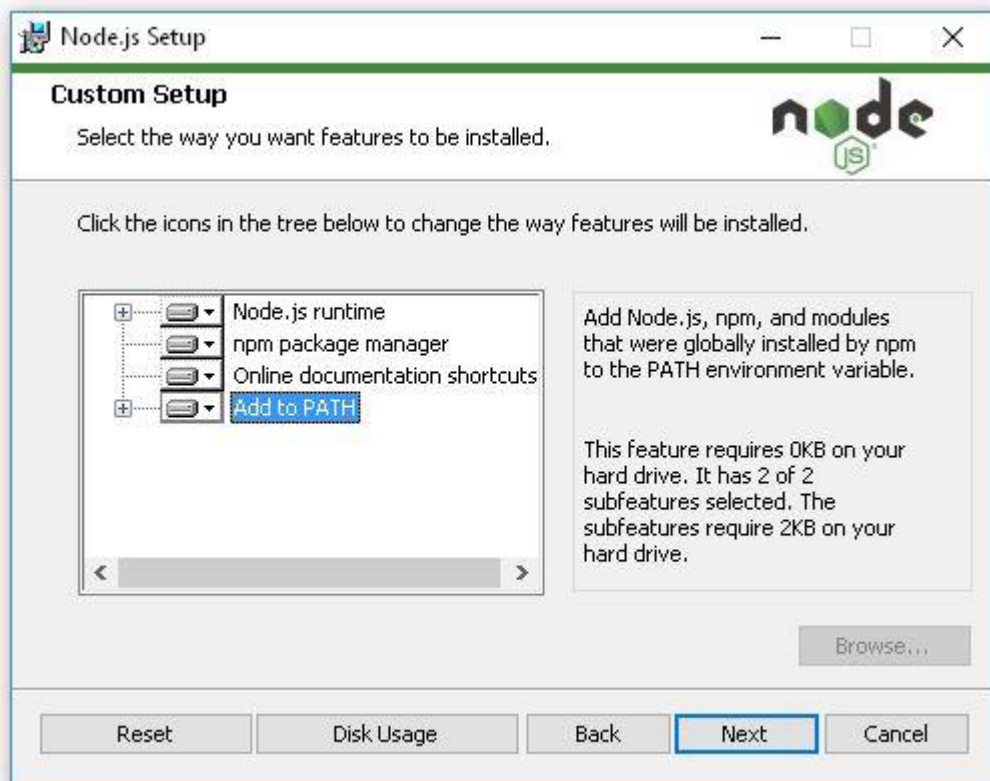
Downloads

Latest LTS Version: **12.18.3** (includes npm 6.14.6)

Download the Node.js source code or a pre-built installer for your platform, and start developing today.

LTS Recommended For Most Users	Current Latest Features	
 Windows Installer <small>node-v12.18.3-x64.msi</small>	 macOS Installer <small>node-v12.18.3.pkg</small>	 Source Code <small>node-v12.18.3.tar.gz</small>
Windows Installer (.msi)	32-bit	64-bit
Windows Binary (.zip)	32-bit	64-bit
macOS Installer (.pkg)	64-bit	
macOS Binary (.tar.gz)	64-bit	
Linux Binaries (x64)	64-bit	

Sau đó chúng ta tiến hành cài đặt như bình thường. Lưu ý trong quá trình cài đặt chúng ta nên chọn để trình Installer gán địa chỉ thư mục vào biến môi trường PATH. Nếu không sau này khi code sẽ rất mệt.



Sau khi cài đặt xong chúng ta kiểm tra lại bằng lệnh `node -v`, nếu thành công máy sẽ hiển thị thông tin version NodeJS được cài đặt

b) Cài đặt Npm

Cài được NodeJS thành công, máy bạn sẽ tự có npm (tức npm đã được cài đặt)

Để kiểm tra phiên bản npm, bạn gõ `npm -v`

c) Cài đặt Angular CLI

CLI là viết tắt command-line interface (interpreter), hiểu đơn giản nó là biến môi trường, khi cài đặt rồi thì window mới hiểu đoạn command đó. Ví dụ bạn gõ `node -v` thì từ khóa `node` được coi là `node CLI`

Để cài đặt Angular CLI thì bạn gõ `npm install -g @angular/cli` trong màn hình command line.

Mẹo: `-g` ở đây là flag global, là tùy chọn toàn cục. Trong npm, khi bạn thêm `-g` khi cài đặt 1 package thì package đó được cài đặt toàn cục.

Để kiểm tra việc cài đặt thành công chưa, bạn gõ `ng -v` trên cửa sổ command line. Nếu hiển thị như bên dưới là bạn đã cài đặt thành công Angular CLI.

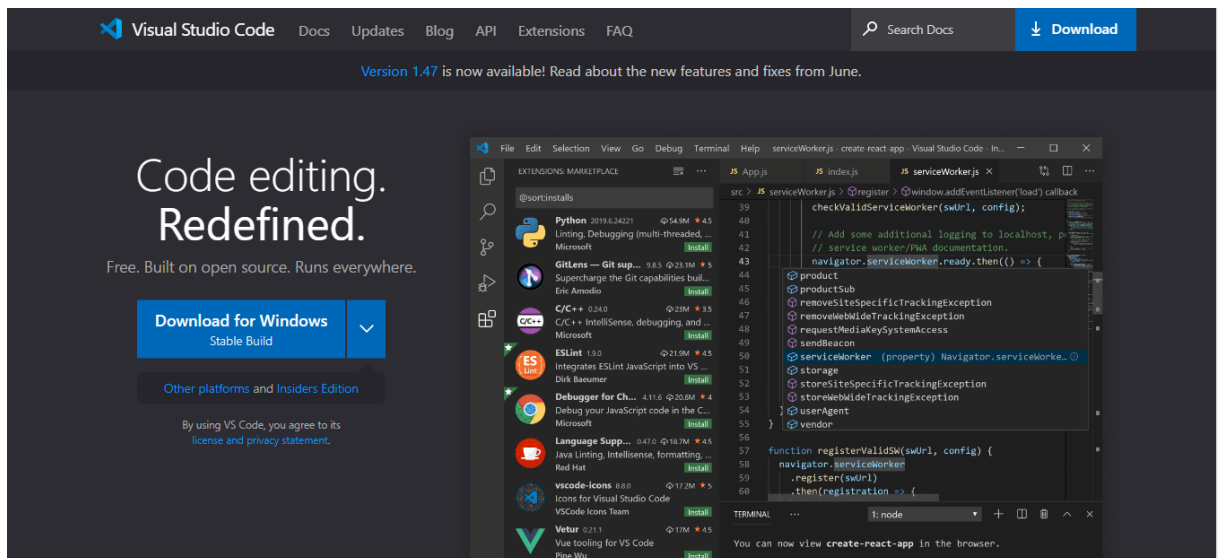


d) Lựa chọn IDE

IDE thì bạn có thể sử dụng IDE tùy thích. Có thể là WebStorm, Atom, VS Code, Netbean, ... hoặc thậm chí là notepad vẫn được. Tuy nhiên thì tốt khuyên khích bạn nên sử dụng VS Code vì nó nhẹ, dễ sử dụng, hiển thị cũng dễ nhìn.

Để cài đặt VS Code

Bạn truy cập trang web <https://code.visualstudio.com/> để tải về bộ cài đặt của VSCode. Chỉ cần nhấn nút Download thì bạn sẽ tải bộ cài phù hợp với hệ thống của mình. Nếu muốn tải những phiên bản khác hoặc cho hệ điều hành khác, bạn chỉ việc nhấn vào nút bên cạnh ô Download và lựa chọn bộ cài mình cần.



Nhấn vào nút download để tải file cài đặt và thực hiện các bước cài đặt theo hướng dẫn

1.3 Tạo project đầu tiên

Sau khi cài đặt xong môi trường phát triển

Để tạo một ứng dụng Angular ta thực hiện thao tác pháp sau

ng new <project-name>

Ví dụ:

Bước 1: Mở terminal (đối với windows: biểu tượng windows + r)

```
Microsoft Windows [Version 10.0.19041.329]
(c) 2020 Microsoft Corporation. All rights reserved.

C:\Users\Nguyen Huu Dong>
```

Bước 2: Chuyển về ổ đĩa và thư mục cần tạo ứng dụng

```
Microsoft Windows [Version 10.0.19041.329]
(c) 2020 Microsoft Corporation. All rights reserved.

C:\Users\Nguyen Huu Dong>cd D:\angular

C:\Users\Nguyen Huu Dong>D:

D:\Angular>
```

Bước 3: Giả sử cần tạo ứng dụng Angular có tên HelloWorld gõ lệnh
ng new HelloWorld

```
Microsoft Windows [Version 10.0.19041.329]
(c) 2020 Microsoft Corporation. All rights reserved.

C:\Users\Nguyen Huu Dong>cd D:\angular

C:\Users\Nguyen Huu Dong>D:

D:\Angular>ng new HelloWorld
```

Đợi trong mười phút hệ thống sẽ tạo ứng dụng xong với tên HelloWorld
Vào đường dẫn đã tạo ở trên để kiểm tra kết quả

Dat (D:) > Angular > HelloWorld

Name	Date modified	Type	Size
e2e	7/26/2020 10:10 AM	File folder	
node_modules	7/26/2020 10:12 AM	File folder	
src	7/26/2020 10:10 AM	File folder	
.browserslistrc	7/26/2020 10:10 AM	BROWSERSLISTRC...	1 KB
.editorconfig	7/26/2020 10:10 AM	Editor Config Sour...	1 KB
.gitignore	7/26/2020 10:10 AM	Text Document	1 KB
angular.json	7/26/2020 10:10 AM	JSON File	4 KB
karma.conf.js	7/26/2020 10:10 AM	JavaScript File	1 KB
package.json	7/26/2020 10:10 AM	JSON File	2 KB
package-lock.json	7/26/2020 10:12 AM	JSON File	510 KB
README.md	7/26/2020 10:10 AM	Markdown Source...	2 KB
tsconfig.app.json	7/26/2020 10:10 AM	JSON File	1 KB
tsconfig.base.json	7/26/2020 10:10 AM	JSON File	1 KB
tsconfig.json	7/26/2020 10:10 AM	JSON File	1 KB
tsconfig.spec.json	7/26/2020 10:10 AM	JSON File	1 KB
tslint.json	7/26/2020 10:10 AM	JSON File	4 KB

Bước 4: Sau khi tạo xong ứng dụng, để chạy ứng dụng ta dùng lệnh sau

ng serve

Trước tiên ta cần chuyển vào thư mục HelloWorld

```
D:\Angular>cd HelloWorld  
D:\Angular\HelloWorld>ng serve
```

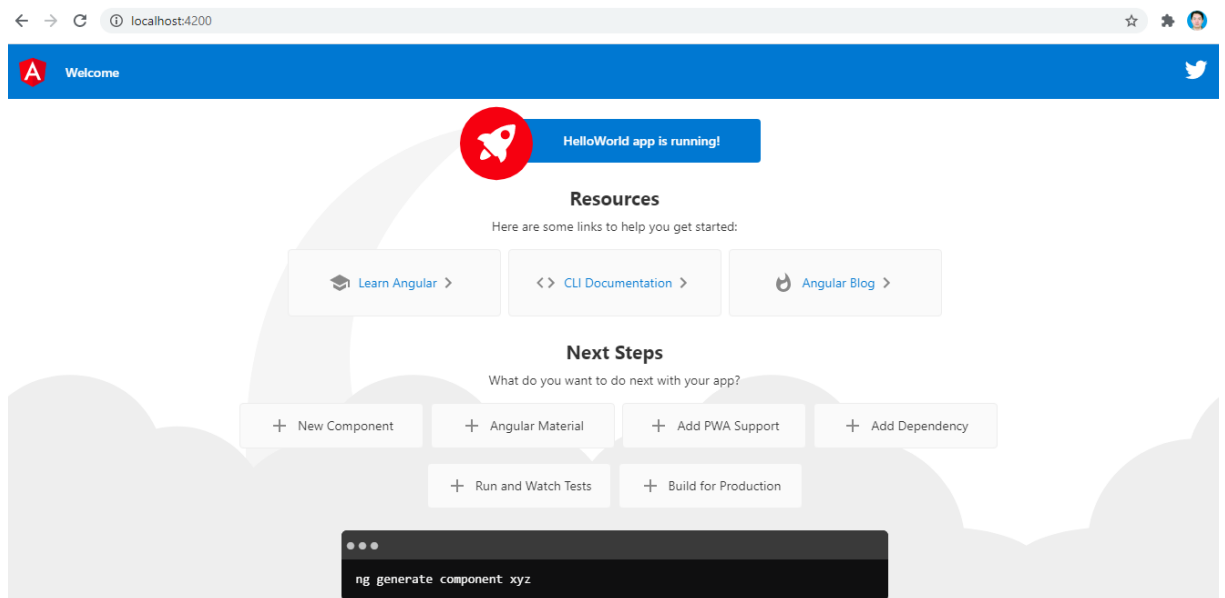
Đội hệ thống dịch

```
D:\Angular\HelloWorld>ng serve  
chunk {main} main.js, main.js.map (main) 60.6 kB [initial] [rendered]  
chunk {polyfills} polyfills.js, polyfills.js.map (polyfills) 141 kB [initial] [rendered]  
chunk {runtime} runtime.js, runtime.js.map (runtime) 6.15 kB [entry] [rendered]  
chunk {styles} styles.js, styles.js.map (styles) 12.4 kB [initial] [rendered]  
chunk {vendor} vendor.js, vendor.js.map (vendor) 2.64 MB [initial] [rendered]  
Date: 2020-07-26T03:22:53.435Z - Hash: a11a3359a78001f05032 - Time: 11215ms  
** Angular Live Development Server is listening on localhost:4200, open your browser on http://localhost:4200/ **  
: Compiled successfully.
```

Bước 5: Khi dịch thành công chúng ta mở một trình duyệt bất kỳ và gõ địa chỉ

<http://localhost:4200>

Kết quả hiển thị

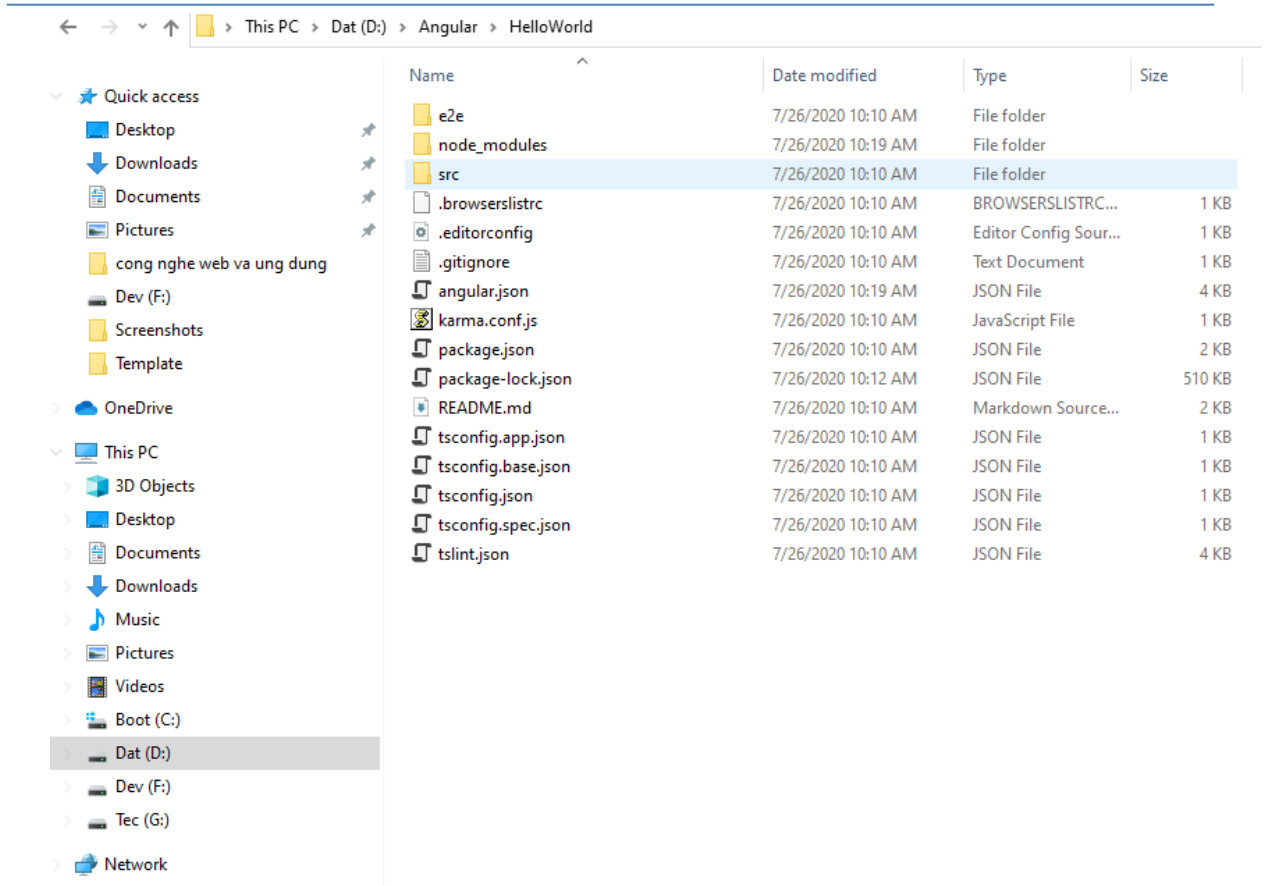


Với màn hình này chúng ta đã tạo và chạy chương trình HelloWorld thành công

Các bạn có thể dùng VS Code để chỉnh sửa lại code và chạy chương trình ngay bên trong đó



Nguyễn Hữu Đông



Cấu trúc thư mục của một angular bao gồm:

- + **e2e**: Thư mục này dùng để chứa các tập tin dành cho mục đích testing.
- + **node_modules**: Các package được cài đặt sử dụng trong project.
- + **src**: Đây là thư mục sẽ chứa toàn bộ source code của ứng dụng Angular.
- + **.editorconfig**: Chứa các cấu hình liên quan đến phần Editor để chỉnh sửa source code như: indent_size, max_line_length,...
- + **.gitignore**: Đây là tập tin metadata của Git, chứa thông tin những tập tin hoặc thư mục sẽ bị ignore không được commit lên Git Repository.
- + **angular.json**: Đây là tập tin chứa cấu hình cho Angular CLI, giúp chúng ta có thể build ứng dụng Angular.
- + **karma.conf.js**: Tập tin cấu hình cho Karma, liên quan nhiều đến phần testing.
- + **package-lock.json**: Dùng để lock version cho các Node.js module dependencies
- + **package.json**: Tập tin cấu hình cho Node.js module dependencies
- + **protractor.conf.js**: Tập tin cấu hình cho Protractor, liên quan nhiều đến phần testing
- + **README.md**: Tập tin này thường được sử dụng để cho các hệ thống Git hiển thị thông tin về Git Repository của chúng ta.

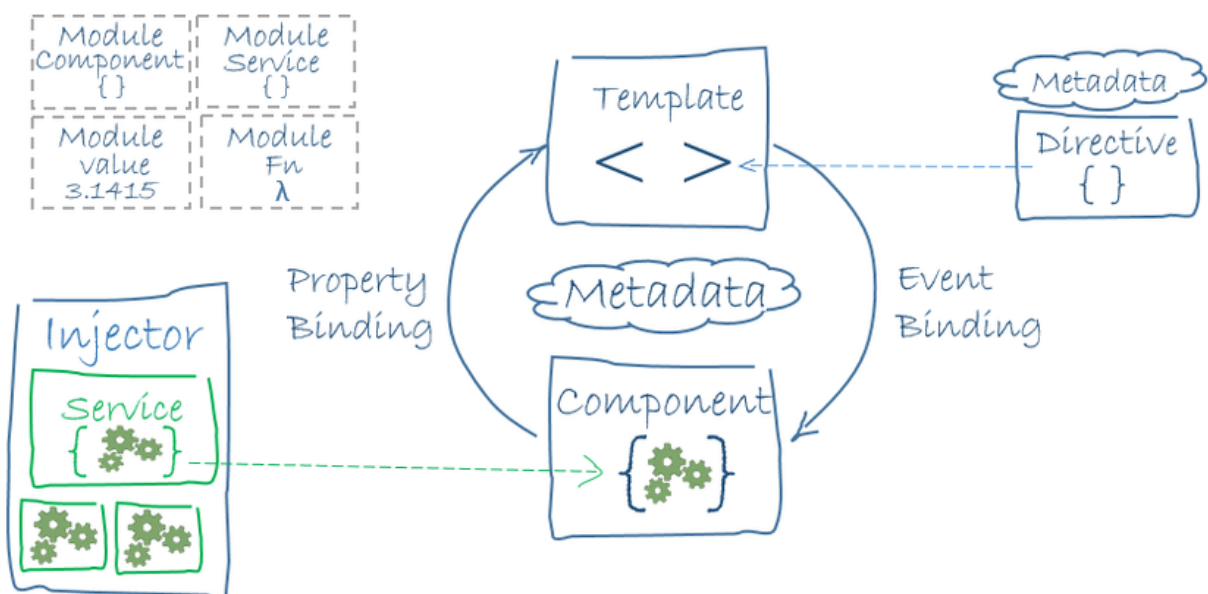
+ **tslint.json**: Tập tin cấu hình để kiểm tra lỗi cho các tập tin .ts (TypeScript) trong Angular project.

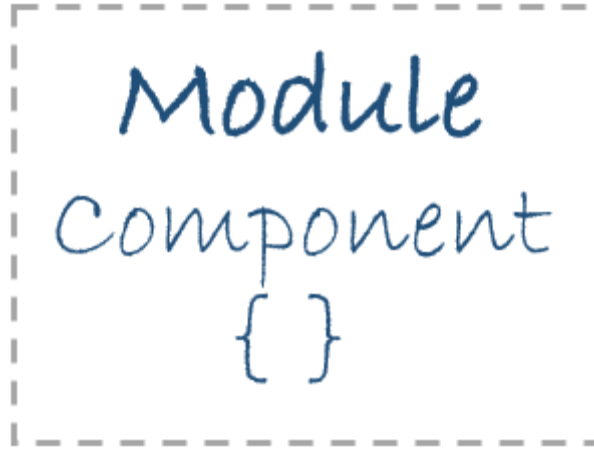
Trong thư mục src có nhưng thư mục sau:

- + **app**: Đây là thư mục sẽ chứa toàn bộ code của ứng dụng Angular.
- + **assets**: Thư mục này sẽ chứa các file ảnh, CSS, custom JavaScript của ứng dụng Angular
- + **environments**: Chúng ta có thể viết ứng dụng chạy trên nhiều môi trường khác nhau, đây chính là thư mục giúp chúng ta làm định nghĩa các tập tin cấu hình cho những môi trường khác nhau đó.
- + **favicon.ico**: Icon của ứng dụng Angular.
- + **index.html**: Trang chủ của ứng dụng Angular.
- + **main.ts**: Chứa code bootstrapping cho ứng dụng Angular.
- + **polyfill.ts**: Dùng để định nghĩa các chuẩn để ứng dụng của chúng ta có thể chạy được trên mọi trình duyệt.
- + **style.css**: Định nghĩa style CSS cho ứng dụng Angular.
- + **test.ts**: Code để chạy test.
- + **tsconfig.json**: Tập tin định nghĩa việc compile cho TypeScript.

1.5 Kiến trúc một ứng dụng Angular

Một ứng dụng Angular được xây dựng từ 8 thành phần sau đây: **Module**, **Component**, **Template**, **Metadata**, **Data Binding**, **Directive**, **Service**, **Dependency Injection**.



*** Module**

Mỗi ứng dụng Angular được gọi là một module và bản thân Angular có riêng một module dùng để quản lý các module khác có tên là Root Module hay NgModule. Root Module thường được đặt tên là AppModule, ngoài root ra thì tùy ứng dụng mà sẽ có thêm các module khác, chúng ta sẽ tìm hiểu về root module trong phần sau.

Chúng ta khai báo một module bằng cách dùng từ khóa `@NgModule`. Các từ khóa như `@NgModule` này là các hàm dùng để chỉnh sửa các lớp của Javascript. Bên trong từ khóa `@NgModule` chúng ta khai báo các tham số sau đây:

- + 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. Đây là một đoạn code module trong file có tên `app.module.ts` đơn giản như sau:

src/app/app.module.ts

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';

@NgModule({
  declarations: [
    AppComponent
```

```

],
imports: [
  BrowserModule,
  AppRoutingModule
],
providers: [],
bootstrap: [AppComponent]
}))
export class AppModule { }

```

Khởi chạy ứng dụng bằng cách bootstrapping root module. Trong quá trình development, chúng ta sẽ bootstrap AppModule trong file main.ts:

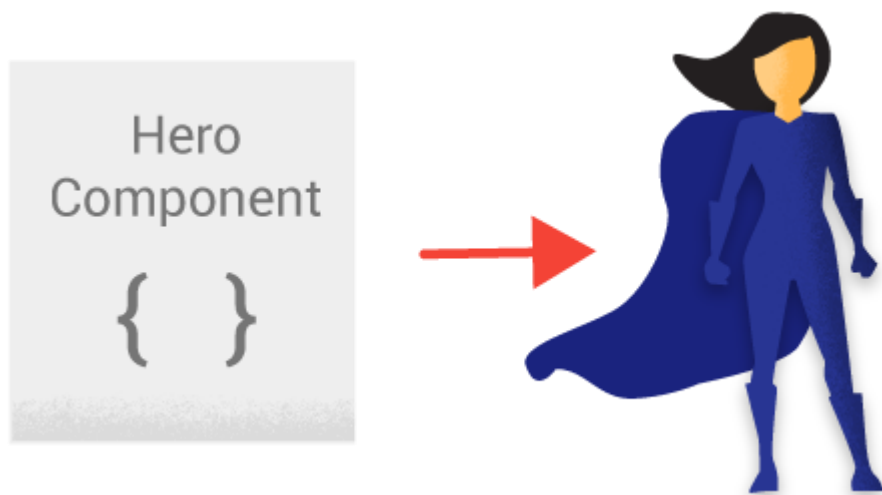
src/main.ts

```

import { enableProdMode } from '@angular/core';
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
import { AppModule } from './app/app.module';
import { environment } from './environments/environment';
if (environment.production) {
  enableProdMode();
}
platformBrowserDynamic().bootstrapModule(AppModule)
  .catch(err => console.error(err));

```

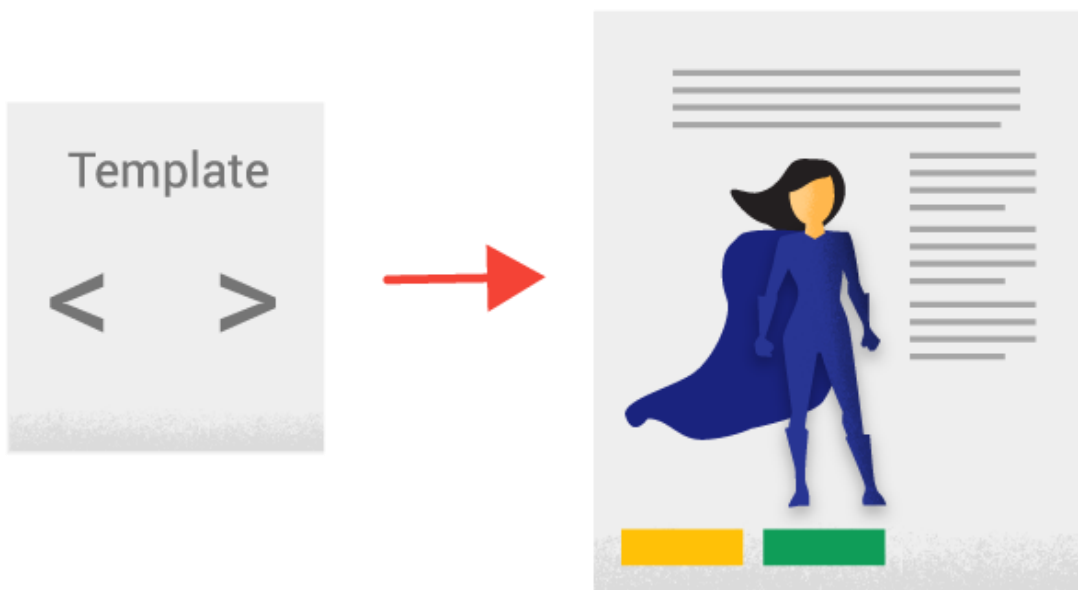
* Component



Component có chức năng điều khiển việc hiển thị, tức là điều khiển View, vậy bạn có thể hình dung Component chính là một Controller trong mô hình MVC...v.v Ví dụ một đoạn code component:

```
export class HeroListComponent implements OnInit {
  heroes: Hero[];
  selectedHero: Hero;
  constructor(private service: HeroService) { }
  ngOnInit() {
    this.heroes = this.service.getHeroes();
  }
  selectHero(hero: Hero) { this.selectedHero = hero; }
}
```

* Template



Template là một đoạn code HTML để component dựa vào đó mà hiển thị trên màn hình. Ví dụ:

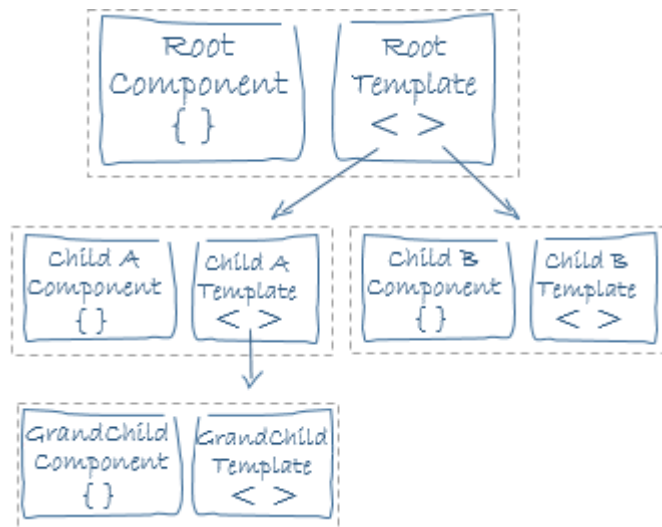
```
<h2>Hero List</h2>
<p><i>Pick a hero from the list</i></p>
<ul>
  <li *ngFor="let hero of heroes" (click)="selectHero(hero)">
    {{hero.name}}
  </li>
```


<hero-detail *ngIf="selectedHero" [hero]="selectedHero"></hero-detail>

Ngoài các thẻ HTML thông thường như <h2>, <p> thì còn có những thẻ và thuộc tính đặc biệt như *ngFor, {{hero.name}}, (click), [hero] và <hero-detail>, đây là cú pháp template của Angular.

Trên dòng cuối cùng của template, <hero-detail> tag là custom element thể hiện cho new component, HeroDetailComponent.

HeroDetailComponent là một component khác với HeroListComponent chúng ta đang xem. HeroDetailComponent mô tả chi tiết từng hero, mà người dùng đã select từ HeroListComponent. HeroDetailComponent là con(child) của HeroListComponent.



* Metadata

Metadata

Metadata (siêu dữ liệu) là những thông tin giúp Angular xử lý các lớp.

Trong đoạn code ví dụ về Component ở trên, đó chỉ là một lớp bình thường viết bằng TypeScript, không có sự xuất hiện của Angular trong này. Muốn Angular hiểu được đó là một lớp dành cho Angular thì chúng ta phải khai báo metadata. Ví dụ:

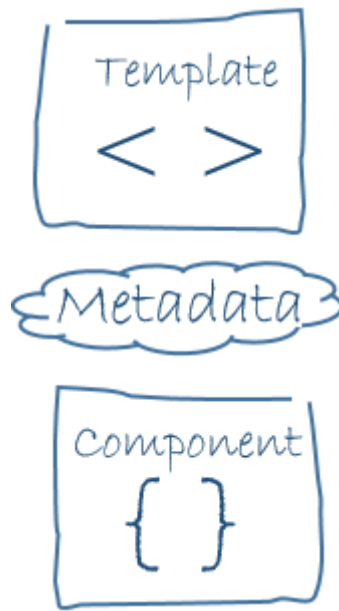
```
@Component({
  moduleId: module.id,
  selector: 'hero-list',
  templateUrl: 'hero-list.component.html',
  providers: [ HeroService ]
})
```

```

}))
export class HeroListComponent implements OnInit {
  /* ... */
}

```

Trong đó `@Component` là từ khóa bắt đầu định nghĩa metadata, phần định nghĩa lớp ngay sau phần metadata này là lớp component của metadata trên. Bên trong chúng ta khai báo một số thông tin cho Angular như `moduleId`, `selector`, `templateUrl`, `providers`. Chúng ta sẽ tìm hiểu về chúng sau.



Metadata trong `@Component` giúp Angular biết cách lấy những thành phần chính tạo nên component. Template, Metadata, và Component được sử dụng cùng với nhau với mục đích tạo nên View. Ngoài `@Component`, chúng ta còn có `@Injectable`, `@Input`, và `@Output` là những decorators rất hay được sử dụng.

* Data Binding

Data Binding tức là lấy dữ liệu từ model/controller đổ vào view. Trong đoạn code ví dụ về template trên có những dòng data binding như sau:

```

<li>{{hero.name}}</li>
<hero-detail [hero]="selectedHero"></hero-detail>
<li (click)="selectHero(hero)"></li>

```

Data binding trong Angular là 2 chiều, tức là chúng ta có thể nhập dữ liệu từ view vào model/controller.

```

<input [(ngModel)]="hero.name">

```

*** Directive**

Directive (chỉ thị) là một lớp và có phần khai báo metadata là @Directive. Thường thì directive sẽ nằm trong một element – hay thẻ của HTML giống như một thuộc tính bình thường.

Có 2 loại directive là structural và attribute.

Các structural directive có chức năng gắn dữ liệu theo một quy tắc nào đó.

```
<li *ngFor="let cus of customer"></li>
<customer *ngIf="selectedCustomer"></customer>
```

Trong đoạn code trên thì *ngFor và *ngIf là các structural directive.

Các attribute directive có chức năng hiển thị dữ liệu một cách trực tiếp.

```
<input [(ngModel)]="hero.name">
```

Trong đoạn code trên thì ngModel là một attribute directive.

*** Service**

Service là các lớp có khả năng thực hiện một số chức năng thường dùng, nói đơn giản thì chúng giống như thư viện vậy. Một số service phổ biến là: logging service, data service, message bus, tax calculator, application configuration.

Ví dụ lớp Logger cho phép chúng ta in các đoạn code báo lỗi, cảnh báo...v.v:

```
@Injectable()
export class Logger {
  log(msg: any) { console.log(msg); }
  error(msg: any) { console.error(msg); }
  warn(msg: any) { console.warn(msg); }
}
```

*** Dependency injection**

Dependency là các lớp/module/service được dùng thêm, Dependency injection là khả năng cho phép tạo các đối tượng lớp có đầy đủ các lớp/module/service được dùng thêm đó. Chẳng hạn như chúng ta có phương thức constructor() như sau:

```
constructor(private service: HeroService) { }
```

Tham số private service: HeroService có nghĩa là lớp này cần dùng một service có tên HeroService.

Angular có riêng một vùng bộ nhớ để lưu trữ các dependency đã được gọi, khi một module/component nào cần dùng service nào, Angular sẽ tìm trong vùng bộ nhớ đó

xem có không, nếu không có thì Angular sẽ tạo một đối tượng của dependency đó và đưa vào bộ nhớ rồi trả về cho lớp đã gọi.

Khi chúng ta xây dựng root module thì chúng ta phải khai báo các dependency trong tham số providers, có như thế Angular mới có thể tìm được.

```
providers: [  
  BackendService,  
  HeroService,  
  Logger  
],
```

Hoặc khai báo ở phần @Component:

```
@Component({  
  moduleId: module.id,  
  selector: 'hero-list',  
  templateUrl: 'hero-list.component.html',  
  providers: [ HeroService ]  
})
```

*** Cách thực thi một ứng dụng Angular**

Thư mục src chứa mã nguồn của ứng dụng và các cài đặt liên quan. Cấu trúc thư mục bao gồm:

app: thư mục chứa module chính của ứng dụng. Chúng ta sẽ tìm hiểu chi tiết về thư mục này trong bài sau.

...

index.html: file chính của chương trình.

Cùng nhìn sơ qua vào nội dung file index.html này:

index.html

```
<!doctype html>  
<html lang="en">  
<head>  
  <meta charset="utf-8">  
  <title>SmartNote</title>  
  <base href="/">  
  <meta name="viewport" content="width=device-width, initial-scale=1">
```

```

<link rel="icon" type="image/x-icon" href="favicon.ico">
</head>
<body>
  <app-root></app-root>
</body>
</html>

```

Bên trong thẻ `<body>` có chứa thẻ `<app-root></app-root>`. Đây là selector được khai báo trong file `app.component.ts`, sử dụng để hiển thị dữ liệu từ file `app.component.html`. Bạn có thể hiểu đơn giản: ta định nghĩa thẻ `<app-root>` "đại diện" cho file `app.component.html`. Như vậy thẻ `<app-root>` đặt ở đâu thì nội dung html của file mà thẻ `<app-root>` này "đại diện" sẽ hiển thị ở đó.

`main.ts`: là file đầu tiên được chạy khi ứng dụng của bạn chạy. Trong này import các module cần thiết cho project. Bạn có thể thấy `angular/core`, `angular/platform-browser-dynamic`, `app.module` và `environment` được import mặc định khi ta tạo project bằng Angular CLI

main.ts

```

import { enableProdMode } from '@angular/core';
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
import { AppModule } from './app/app.module';
import { environment } from './environments/environment';
if (environment.production) {
  enableProdMode();
}
platformBrowserDynamic().bootstrapModule(AppModule)
  .catch(err => console.log(err));

```

Khi dòng lệnh

```
platformBrowserDynamic().bootstrapModule(AppModule)
```

được thực thi sẽ gọi đến `AppModule`. `AppModule` được khai báo trong file `app.module`. Trong file `app.module` có sử dụng `AppComponent` trong file `app.component`. Trong `app.component` lại khai báo selector `<app-root>`. Thẻ `<app-root>` được gọi ở file `index.html`. Vậy file `index.html` được thực thi ở trình duyệt

Như vậy ta có luồng thực thi của ứng dụng như sau:

main.ts -> app.module.ts -> app.component.ts -> index.html

1.6. Một số câu lệnh cơ bản của Angular Cli

Angular CLI, hay Angular Command Line Interface, giúp chúng ta dễ dàng bắt đầu với bất kì project Angular nào. Với Angular CLI, chúng ta có thể tạo ra các component, pipe, directive,... nhanh chóng bằng cách chỉ sử dụng dòng lệnh.

+ Chạy chương trình :

- ng serve hoặc ng serve –open: Biên dịch project, chạy ở cổng 4200 và tự động mở lên trình duyệt khi biên dịch xong
- ng serve --port 3006 –open: Biên dịch project, chạy ở cổng 3006 và tự động mở lên trình duyệt khi biên dịch xong

+ Dịch chương trình : Các bạn sử dụng câu lệnh : ng build --prod

+ Test chương trình: Các bạn sử dụng câu lệnh : ng test

+ Kiểm tra cú pháp TypeScript : Các bạn sử dụng câu lệnh : ng lint

Các câu lệnh này đều được mô tả trong file package.json

Cài đặt các gói được mô tả trong package.json: npm install hoặc npm i

Các câu lệnh thường dùng khác:

+ Tạo Component : Các bạn sử dụng câu lệnh : ng g component my-new-component

+ Tạo Service : Các bạn sử dụng câu lệnh : ng g service my-new-service

+ Tạo Module : Các bạn sử dụng câu lệnh : ng g module my-module

+Tạo Class: Các bạn sử dụng câu lệnh : ng g class my-new-class

+Tạo Interface: Các bạn sử dụng câu lệnh : ng g class my-new-class

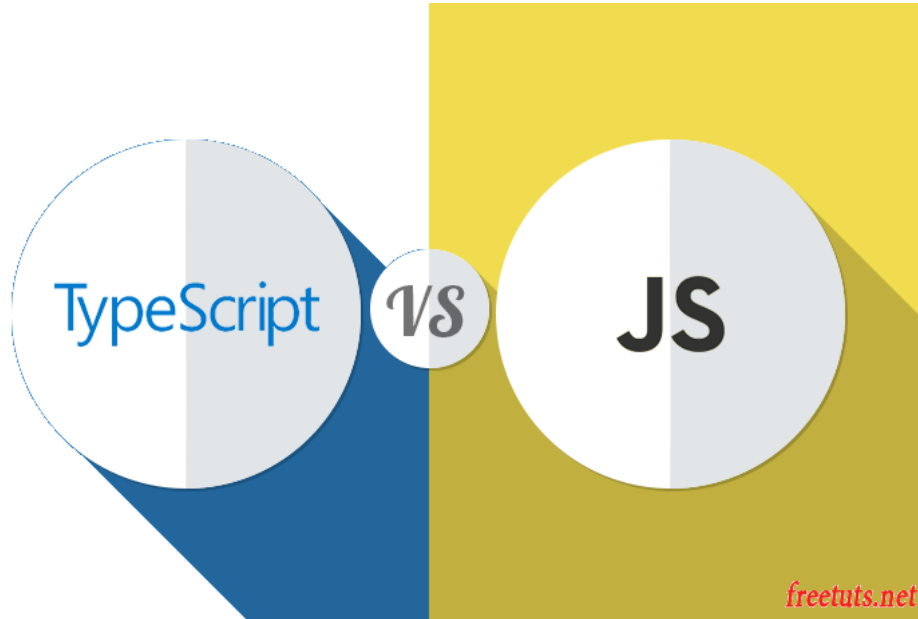
+Tạo Enum: Các bạn sử dụng câu lệnh : ng g enum my-new-enum

+Tạo Pipe: Các bạn sử dụng câu lệnh : ng g pipe my-new-pipe

Bài 2: Cơ bản về ngôn ngữ lập trình TypeScript

2.1 TypeScript là gì?

TypeScript là một dự án mã nguồn mở được phát triển bởi Microsoft, nó có thể được coi là một phiên bản nâng cao của Javascript bởi việc bổ sung tùy chọn kiểu tĩnh và lớp hướng đối tượng mà điều này không có ở Javascript. TypeScript có thể sử dụng để phát triển các ứng dụng chạy ở client-side (*Angular2*) và server-side (*NodeJS*).



TypeScript sử dụng tất cả các tính năng của của ECMAScript 2015 (ES6) như classes, modules. Không dừng lại ở đó nếu như ECMAScript 2017 ra đời thì mình tin chắc rằng TypeScript cũng sẽ nâng cấp phiên bản của mình lên để sử dụng mọi kỹ thuật mới nhất từ ECMAScript. Thực ra TypeScript không phải ra đời đầu tiên mà trước đây cũng có một số thư viện như CoffeScript và Dart được phát triển bởi Google, tuy nhiên điểm yếu là hai thư viện này sử dụng cú pháp mới hoàn toàn, điều này khác hoàn toàn với TypeScript, vì vậy tuy ra đời sau nhưng TypeScript vẫn đang nhận được sự đón nhận từ các lập trình viên.

2.2 Tại sao nên sử dụng Typescript

Để hiểu tại sao nên sử dụng TypeScript thì có lẽ chúng ta nên tìm hiểu sơ lược về các ưu điểm mà Typescript mang lại.

Dễ phát triển dự án lớn: Với việc sử dụng các kỹ thuật mới nhất và lập trình hướng đối tượng nên TypeScript giúp chúng ta phát triển các dự án lớn một cách dễ dàng.

Nhiều Framework lựa chọn: Hiện nay các Javascript Framework đã dần khuyến khích nên sử dụng TypeScript để phát triển, ví dụ như Angular và Ionic

Hỗ trợ các tính năng của Javascript phiên bản mới nhất: TypeScript luôn đảm bảo việc sử dụng đầy đủ các kỹ thuật mới nhất của Javascript, ví dụ như version hiện tại là ECMAScript 2015 (ES6).

Là mã nguồn mở: TypeScript là một mã nguồn mở nên bạn hoàn toàn có thể sử dụng mà không mất phí, bên cạnh đó còn được cộng đồng hỗ trợ.

TypeScript là Javascript: Bản chất của TypeScript là biên dịch tạo ra các đoạn mã javascript nên bạn có thể chạy bất kỳ ở đâu miễn ở đó có hỗ trợ biên dịch Javascript. Ngoài ra bạn có thể sử dụng **trộn lẫn cú pháp của Javascript vào bên trong TypeScript**, điều này giúp các lập trình viên tiếp cận TypeScript dễ dàng hơn.

Như vậy bản chất của TypeScript là một trình biên dịch xuất ra mã Javascript dựa vào cấu trúc riêng của nó. Trong CSS cũng có một thư viện tương tự đó là SASS và LESS.

Như vậy ta có thể coi TypeScript là cha của Javascript bởi kết quả sau khi biên dịch TypeScript là xuất ra các đoạn mã Javascript, để các bạn hiểu rõ hơn mình sẽ lấy một ví dụ như sau:

Code TypeScript

```
class Customer {  
    Name : string;  
    constructor (firstName: string, lastName: string)  
    {  
        this.Name = firstName + " " + lastName;  
    }  
    GetName()  
    {  
        return "Hello, " + this.Name;  
    }  
}
```

Biên dịch thành Javascript

```
var Customer = (function () {  
    function Customer(firstName, lastName) {  
        this.Name = firstName + " " + lastName;  
    }  
    Customer.prototype.GetName = function () {
```

```

    return "Hello, " + this.Name;
};
return Customer;
}());

```

Với hai đoạn code trên thì rõ ràng nhìn vào mã của TypeScript rất là trong sáng và mạch lạc.

2.3 Các kiểu dữ liệu cơ bản và cách khai báo biến

TypeScript là một ngôn ngữ có “họ hàng” với Javascript. Đây là một bản mở rộng của ECMAScript6 (viết tắt là ES6), ES6 lại là một bản mở rộng của ECMAScript5 (viết tắt là ES5), ES5 chính là Javascript mà chúng ta thường dùng.

2.3.1 Các kiểu dữ liệu cơ bản

Bản chất của Typescript vẫn là Javascript nên các kiểu dữ liệu cơ bản của Javascript thì Typescript đều có ngoài ra Typescript còn có một số kiểu dữ liệu khác như là enum, tuple, any, void ...

Các kiểu dữ liệu cơ bản

Các kiểu dữ liệu thường dùng là string, number, boolean, array, enum, any, void.

a) Kiểu string

Kiểu string đơn giản là một chuỗi (một đoạn text) được bao bọc bởi cặp ký tự ' hoặc "

Ví dụ 1:

```

var string1 : string;
string1 = '1001';
console.log(string1);

```

Ví dụ 2:

```

let employeeName:string = "John Smith";
let employeeDept:string = "Finance";

```

b) Kiểu number

Cũng tương tự như trong Javascript thì trong TypeScript chỉ tồn tại một kiểu Number, còn việc phân chia các kiểu nhỏ hơn như số nguyên, số thực sẽ phụ thuộc vào giá trị mà bạn gán cho nó. TypeScript cũng hỗ trợ kiểu nhị phân và bát phân được giới thiệu trong ECMAScript 2015.

Ví dụ:

```

var decimal: number = 12;

```

```
var hex: number = 0xf00d;  
var binary: number = 0b1010;  
var octal: number = 0o744;
```

c) Kiểu Array

Kiểu mảng, khi khai báo các phần tử trong mảng thì chúng ta phải chỉ ra cả kiểu dữ liệu nữa. Để tạo array thì chúng ta có thể dùng cú pháp `Array<kiểu>` hoặc `<kiểu>[]`.

Ví dụ:

Mảng string

```
var arrString : string[];  
arrString = ['teo', 'ty', 'tun'];  
console.log(arrString[0]);
```

Mảng number

```
var arrNumber : number[];  
arrNumber = [1, 2, 3];
```

d) Kiểu boolean

Kiểu boolean có hai giá trị là `true` hoặc `false` và cả Javascript lẫn TypeScript đều gọi là `boolean`.

Ví dụ:

```
var boolean1 : boolean = true;  
console.log(boolean1);
```

e) Kiểu enum

Tương tự như trong C#, Enum là kiểu dữ liệu đặc biệt dùng để tạo một nhóm tên tương ứng với các giá trị là những con số mà ta thiết lập cho nó, cách này sẽ giúp ta dễ dàng nhớ tên hơn.

Ví dụ:

```
enum Color {Red, Green, Blue}  
var c: Color = Color.Green;
```

f) Kiểu tuple

Tuple là kiểu dữ liệu đặc biệt có thể chứa nhiều giá trị với nhiều kiểu dữ liệu con khác nhau. Thực ra Tuple là một mảng nhưng đã xác định được số phần tử và kiểu dữ liệu cho mỗi phần tử đó.

Ví dụ:

```
var x: [string, number];  
x = ['ahihi', 10];  
for (let i = 0; i < x.length; i++) {  
    console.log(x[i]);  
}  
console.log(x[0]);
```

g) Kiểu any

Đây là kiểu dữ liệu thoải mái nhất bởi nó cho phép bạn gán giá trị với kiểu dữ liệu bất kì, điều này giúp giải quyết rắc rối ở một số trường hợp, ví dụ ta cần lấy dữ liệu từ người dùng hoặc một thư viện khác thì ta không biết giá trị trả về sẽ ở kiểu dữ liệu nào nên ta sẽ sử dụng kiểu Any để tránh lỗi. Sau đây là một ví dụ từ trang chủ của nó.

Ví dụ:

```
var something: any = "this is a string";  
something = 1;  
something = [1, 2, 3];
```

h) Kiểu void

Trong C# thì khi muốn khai báo một hàm không có giá trị trả về thì ta sẽ sử dụng hàm void, tuy ta hay gọi là hàm void nhưng thực ra nó là một kiểu dữ liệu với giá trị là null, trong TypeScript thì có thêm giá trị undefined.

Ví dụ:

```
function showMessage(): void {  
    alert("Success!");  
}  
let unusable: void = undefined;
```

2.3.2 Cách khai báo biến

Ngoài từ khóa var ra TypeScript hỗ trợ thêm hai cách khai báo biến đó là sử dụng từ khóa let và const.

a) Khai báo biến với từ khóa var

Ví dụ:

```
var domain = 'freetuts.net';  
var author = 'Nguyễn Văn Cường';  
var series = new Array();
```


Sử dụng từ khóa `var` đồng nghĩa với việc bạn đã khai báo trong phạm vi toàn cầu, nghĩa là bạn có thể sử dụng nó ở mọi nơi từ bên trong hàm cho đến bên ngoài hàm.

Ví dụ:

```
var domain = 'utehy.edu.vn';  
  
function showDomain()  
{  
    alert(domain);  
}  
  
// Kết quả: utehy.edu.vn  
  
showDomain();
```

Nếu bạn khai báo một biến nằm bên trong hàm thì đương nhiên biến đó chỉ mang tính chất là biến cục bộ, nghĩa là chỉ sử dụng được trong hàm đó mà thôi.

Ví dụ:

```
function showDomain()  
{  
    var domain = 'utehy.edu.vn';  
}  
  
// Lỗi vì biến domain là biến cục bộ trong hàm showDomain  
alert(domain);
```

b) Khai báo biến với từ khóa `let`

Từ khóa `let` được giới thiệu ở phiên bản ECMAScript 6 (ES6). Khi sử dụng từ khóa `let` để khai báo biến thì biến đó chỉ hoạt động trong phạm vi khối của nó (block-scoped).

Ví dụ:

```
let a = 20;  
let b = 30;  
let c = a + b;  
alert(c); // 50
```

Nếu sử dụng bên ngoài phạm vi của block-scoped thì sẽ bị lỗi, xem ví dụ.

Ví dụ:

```
let domain = 'utehy.edu.vn';  
if (domain == 'utehy.edu.vn')
```

```
{
  let author = 'Nguyễn Văn Anh';
}
alert(author); // Lỗi vì biến author nằm trong phạm vi block-scoped khác
```

c) Khai báo biến với từ khóa const

Sử dụng từ khóa const để khai báo một biến và biến đó sẽ không thể nào thay đổi giá trị lại, trong các ngôn ngữ lập trình khác thì đây cũng có thể hiểu là khai báo hằng. Khác với các ngôn ngữ khác là khi khai báo hằng thì thường ta chỉ gán được kiểu giá trị là kiểu chuỗi hoặc kiểu số, tuy nhiên trong Javascript nói chung và trong TypeScript nói riêng thì bạn có thể gán cho nó mọi kiểu dữ liệu.

Như ví dụ sau là hoàn toàn hợp lệ vì ta gán giá trị là một object.

```
const info = {
  name : "Nguyen Van Cuong",
  domain : "Freetuts.net"
};
console.log(info);
```

Nếu bạn cố tình thay đổi giá trị thì sẽ bị báo lỗi.

```
// Khai báo const
const domain = 'utehy.edu.vn';
// Thay đổi giá trị => dòng này sẽ bị lỗi
domain = 'fit.utehy.edu.vn ';
```

Giống như let, phạm vi hoạt động của biến const cũng bị hạn chế bởi block-scoped.

```
{
  // Khai báo const
  const domain = 'utehy.edu.vn';
  console.log(domain);
}
{
  console.log(domain); // Bị lỗi vì biến domain nằm ở block-scoped phía trên
}
```

2.4 Các cấu trúc điều khiển

2.4.1 Lệnh If Else trong TypeScript

Lệnh if dùng để kiểm tra một biểu thức đúng hay sai? Nếu đúng thì thực thi một nhiệm vụ nào đó.

```
if (condition)
{
    // thực hiện
}
```

Trong đó condition là biểu thức muốn kiểm tra, nó có thể là một biểu thức, một giá trị hoặc là một biến.

Ví dụ: Kiểm tra nếu điểm bé hơn 5 thì thông báo thi rớt.

```
var point = 4;
if (point < 5)
{
    alert('Rớt');
}
```

Lệnh else sẽ được thực hiện nếu như biểu thức ở lệnh if không thỏa điều kiện.

```
if (condition){
    // Do Something
}
else{
    // Do Something
}
```

Ví dụ: Kiểm tra điểm lớn hơn 4 thì thông báo đậu, ngược lại thông báo rớt.

```
var point = 2;
if (point >= 5){
    alert('Đậu');
}
else{
    alert('Rớt');
}
```

Lệnh else if dùng để kiểm tra thêm một điều kiện và nó sẽ được chạy nếu tất cả các lệnh kiểm tra điều kiện ở trên nó không thỏa.

```
if (condition1){
```

```
}  
else if (condition2){  
}  
else if (condition3){  
}  
else{  
}
```

Ví dụ: Kiểm tra các mức điểm của học sinh

```
var point = 10;  
if (point >= 8){  
    alert('Giỏi');  
}  
else if (point >= 6){  
    alert('Khá');  
}  
else if (point >= 5){  
    alert('Trung bình');  
}  
else{  
    alert('Yếu');  
}
```

2.4.2 Lệnh Switch Case trong TypeScript

Lệnh Switch Case có chức năng tương tự như if else, tuy nhiên nó chỉ dùng để kiểm tra biểu thức có giá trị là một chuỗi hoặc một con số.

Cú pháp như sau:

```
switch (expression) {  
    case case1:  
        statements1  
        [break;]  
    case case2:  
        statements2  
        [break;]
```

```
case case3:
    statements3
    [break;]
default:
    default statements
    [break;]
}
```

Trong đó lệnh break sẽ có nhiệm vụ dừng chương trình switch case.

Ví dụ:

```
var book = 'Math';
var msg = "";
switch (book) {
    case "English":
        msg = "Sách có giá là $12.";
        break;
    case "Math":
        msg = "Sách có giá là $22.";
        break;
    case "Commerce":
        msg = "Sách có giá là $12.";
        break;
    case "History":
        msg = "Sách có giá là $125.";
        break;
    case "Physics":
        msg = "Sách có giá là $12.99.";
        break;
    default:
        msg = "Không tìm thấy mệnh giá";
}
alert(msg);
```

Trong ví dụ này vì cuốn sách có tên là Math nên case "Math" sẽ được chạy, lúc này biến msg sẽ có giá trị là "Sách có giá là \$22." và tiếp theo lệnh break; sẽ dừng chương trình switch lại.

2.4.3 Vòng lặp trong TypeScript

Vòng lặp rất quan trọng trong các ngôn ngữ lập trình và trong TypeScript cũng vậy. Khi nói đến vòng lặp thì ta sẽ nhớ đến các vòng lặp thông dụng như: vòng lặp for, vòng lặp while vòng lặp do while. Tuy nhiên với Javascript thì chúng ta sẽ có thêm một số vòng lặp khác nữa như vòng lặp for in và vòng lặp for as.

a) Vòng lặp for in

Vòng lặp for in ta còn gọi là iterates, tức là sau mỗi lần lặp nó sẽ nhớ là đang lặp tới phần tử nào để lần lặp tiếp theo nó tự động lấy phần tử tiếp theo. Giá trị của mỗi lần lặp là index chứ không phải là value.

Cú pháp:

```
for (variable in object)
{
    //block to execute
}
```

Ví dụ:

```
var list = [1,2,5];
var t = 0;
for (var i in list) {
    t += list[i];
}
```

Trong vòng lặp này thì mỗi lần lặp nó sẽ lặp một phần tử với i lần lượt là chỉ số của mảng: 0,1,2.

b) Vòng lặp for

Vòng lặp for dùng để lặp những trường hợp ta biết tổng số lần lặp, xem thêm tại bài vòng lặp for trong javascript.

Ví dụ1:

```
for (var j = 0; j <= 10; j++)
{
    document.write(j);
}
```

```
}
```

Ví dụ 2:

```
var numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];  
var sum = 0;  
for(var i = 0; i < numbers.length; i++)  
    sum += numbers[i];  
console.log(sum);
```

c) Vòng lặp while và do while

Khác với vòng lặp for, vòng lặp while và do while dùng để lặp cho trường hợp ta chưa biết tổng số lần lặp. Riêng đối với vòng lặp do while thì nó sẽ luôn luôn lặp ít nhất một lần. Xem thêm tại vòng lặp while và do while trong Javascript.

Vòng lặp while

Cú pháp:

```
while (variable<=endvalue) {  
    //code block to be executed  
}
```

Ví dụ:

```
var i = 10;  
while (i <= 1000) {  
    console.log(i);  
    i++;  
}
```

Vòng lặp do while

Cú pháp:

```
do{  
    // do something  
}  
while (variable<=endvalue);
```

Ví dụ:

```
var i = 10;s  
do {  
    console.log(i);
```

```

    i++;
  } while (i <= 1000);

```

d) Vòng lặp for of

Về bản chất vòng lặp for of giống vòng lặp for in, tuy nhiên điểm khác biệt duy nhất là vòng lặp for of sẽ trả về value chứ không phải là index.

Ví dụ:

```

let numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
var s = "";
var o = 0;
for (let num of numbers) {
    s = s + "\n Array Value is - " + num;
}
alert(s);

```

2.5 Hàm và cách khai báo

2.5.1 Khai báo

Giống như trong Javascript(Js) hàm trong Typescript(Ts) có cú pháp khai báo cũng rất đơn giản :

- Name function:

```

function add(x: number, y: number): number {
    return x + y;
};

```

- Anonymous function:

```

let myAdd = function(x: number, y: number): number {
    return x + y;
};

```

Bên trên là 2 cách khai báo function cơ bản nhất trong TypeScript :

+ Chúng ta bắt đầu với từ ‘function’ quen thuộc rồi đến ‘add’ (tên function) và bên trong dấu ngoặc đơn sẽ là các parameters(tham số). Đến đây chúng ta sẽ nhận thấy điều khác biệt lớn nhất trong cách khai báo hàm giữa Js và Ts là chúng ta sẽ khai báo cả kiểu cho tham số. Tiếp đến là chúng ta sẽ khai báo kiểu trả về cho hàm. Và cuối cùng là chúng ta sẽ viết code logic vào phần thân hàm.

+ Typescript có thể tìm ra kiểu trả về thông qua các câu lệnh return, vì vậy chúng ta có thể tùy ý bỏ qua khai báo kiểu trả về trong nhiều trường hợp.

2.5.2 Function type (kiểu hàm):

Bạn có thể khai báo một biến có kiểu là một hàm như sau:

```
let varFun: (hello: number, word: number) => number = function(x: number, y:
number): number {
    return x + y;
};
```

Một function type có 2 phần. Kiểu của các đối số về kiểu trả về. 2 phần này là bắt buộc khi chúng ta muốn khai báo một biến với function type

- + Phần đầu là mình khai báo kiểu và số lượng tham số, xếp theo thứ tự tham số chuyển vào hàm tương ứng.
- + Tiếp theo là phần chỉ ra kiểu trả về của hàm đó, bằng cách sử dụng ‘=>’ ngăn giữa các tham số và loại trả về. Vì đây là phần bắt buộc nên khi hàm không trả về gì thì ta sẽ điền là ‘void’ thay vì bỏ đi.

2.5.3 Inferring the type (suy kiểu)

Khi bạn khai báo một biến với function type như sau:

```
let varFun: (hello: number, word: string) => number = function(x, y): number {
    return x + y;
};
```

Trình biên dịch sẽ tự động suy ra kiểu của các tham số x, y lần lượt là number và string.

Điều này được gọi là ‘contextual typing’ (suy kiểu theo ngữ cảnh). Nó giúp chúng ta giảm bớt công gõ code.

2.5.4 Optional and default parameter (tham số tùy chọn và mặc định)

- Khi gọi một hàm thì ta phải chắc chắn số lượng đối số được cung cấp phải khớp với số lượng tham số mà hàm mong đợi. Ví dụ như:

```
function setName(nameOne: string, nameTwo: string): string {
    return nameOne + nameTwo
}

const result1 = setName('superman') // error, too few parameters
```

```
const result2 = setName('ironman', 'spiderman', 'batman') // error, too many
parameters
```

```
const result3 = setName('hello', 'word' ) // hello word
```

- Trong Javascript mọi tham số là tùy chọn và họ có thể bỏ qua một trong số chúng nếu họ muốn. Chúng ta cũng có thể làm điều này với typescript bằng cách thêm ‘?’ vào sau tham số mà chúng ta muốn nó là tùy chọn. Ví dụ như:

```
function setName(nameOne: string, nameTwo?: string): string {
    if(nameTwo) return nameOne + nameTwo;
    else return nameOne;
}
```

```
const result1 = setName('superman') // superman
```

```
const result2 = setName('ironman', 'spiderman', 'batman') // error, too many
parameters
```

```
const result3 = setName('hello', 'word' ) // hello word
```

Lưu ý: Trong hàm thì tham số tùy chọn không thể đứng đầu mà nó chỉ có thể viết sau tham số bắt buộc.

- Trong Typescript, chúng ta có thể set một giá trị cho một tham số khi người dùng không cung cấp. Chúng được gọi là tham số khởi tạo mặc định. Ví dụ như :

```
function setName(nameOne: string, nameTwo = "man"): string {
    return nameOne + " " + nameTwo;
}
```

```
const result1 = setName('super') // super man
```

```
const result2 = setName('ironman', 'spiderman', 'batman') // error, too many
parameters
```

```
const result3 = setName('hello', 'word' ) // hello word
```

```
const result4 = setName('wonder', undefined) // wonder man
```

Lưu ý: Cũng giống như các tham số tùy chọn, vị trí của tham số mặc định phải nên được viết sau tất cả các tham số bắt buộc. Tuy nhiên nếu chúng ta vẫn muốn viết chúng trước các tham số bắt buộc thì khi gọi hàm đó chúng ta một là phải truyền giá trị cho chúng, hai là vượt qua dùng giá trị mặc định bằng cách truyền ‘undefined’.

2.4.5 Rest Parameter

Đôi khi bạn sẽ muốn làm việc với rất nhiều tham số trong một kiểu. Hay là bạn không biết phải nên có bao nhiêu tham số cho hàm này.

Trong TypeScript chúng ta sẽ tập hợp những tham số đó lại thành 1 biến. Ví dụ như:

```
function setName(nameOne: string,... arrayNameTwo: string[]): string {  
    return nameOne + " " + arrayNameTwo.join("");  
}
```

=> Khi đó chúng ta có thể truyền bao nhiêu tham số tùy thích.

```
let result = setName("ironman", "superman", "batman", "spiderman").
```

Rest parameter được coi là vô số các tham số tùy chọn. Khi truyền đối số cho rest parameter thì ta có thể truyền bao nhiêu tùy ý. Trình biên dịch sẽ tạo thành 1 mảng các đối số truyền vào với tên được đặt sau dấu (...), cho phép bạn sử dụng nó trong hàm của mình.

2.5.6 Arrow function

a) Khai báo

Ví dụ khai báo arrow function như sau:

```
var hello = (name, message) => {  
    console.log("Chào " + name + ", bạn là " + message);  
};  
hello('Cường', 'admin freetuts.net');
```

Cách viết thường

```
function hello(name, message)  
{  
    console.log("Chào " + name + ", bạn là " + message);  
}  
hello('Cường', 'admin freetuts.net');
```

So sánh hai cách trên thì rõ ràng cách thông thường sẽ đơn giản hơn rất nhiều, và cả hai đoạn code đều cho kết quả như sau:

Nội dung là một câu lệnh đơn:

Trường hợp trong thân của hàm chỉ có một lệnh duy nhất thì bạn có thể sử dụng theo ví dụ dưới đây.

```
var hello = (name, message) => console.log("Chào " + name + ", bạn là " +  
message);
```

Nghĩa là bạn có thể bỏ đi cặp dấu {}, điều này tuân thủ theo nguyên tắc "nếu bên thân cặp {} chỉ là một câu lệnh thì bạn có thể bỏ cặp {}".

Trường hợp một tham số:

Trường hợp truyền vào chỉ một tham số thì bạn có thể bỏ cặp ().

```
var hello = message => {
  console.log(message);
};
hello('Chào mừng bạn đến với utehy');
```

Trường hợp không có tham số:

Trường hợp không có tham số truyền vào thì bạn sử dụng cặp () rỗng, xem ví dụ sau:

```
var hello = () => {
  console.log('Chào mừng bạn đến với utehyt');
};
hello();
```

2.6 Khám phá chi tiết một số kiểu dữ liệu

2.6.1 Kiểu dữ liệu string

Cách khai báo

```
let employeeName1:string;
let employeeName2:string = 'John Smith';
let employeeName3:string = "John Smith";
```

Chuỗi là một kiểu dữ liệu nguyên thủy khác được sử dụng để lưu trữ dữ liệu văn bản. Các giá trị chuỗi được bao quanh bởi dấu ngoặc kép đơn hoặc dấu ngoặc kép.

Kể từ phiên bản TypeScript 1.4, TypeScript đã bao gồm hỗ trợ cho các chuỗi Mẫu ES6. Chuỗi mẫu được sử dụng để nhúng biểu thức vào chuỗi.

```
let employeeName:string = "John Smith";
let employeeDept:string = "Finance";
// Pre-ES6
let employeeDesc1: string = employeeName + " works in the " + employeeDept +
" department.";
// Post-ES6
```

```
let employeeDesc2: string = `${employeeName} works in the ${employeeDept}
department.`;
```

```
console.log(employeeDesc1);//John Smith works in the Finance department.
```

```
console.log(employeeDesc2);//John Smith works in the Finance department.
```

Ở đây, thay vì viết một chuỗi là sự kết hợp giữa văn bản và các biến với các phép nối, chúng ta có thể sử dụng một câu lệnh với dấu back-tick ` . Các giá trị biến được viết là . Sử dụng các chuỗi mẫu, việc nhúng các biểu thức sẽ dễ dàng hơn và cũng ít tẻ nhạt hơn khi viết các chuỗi dựa trên văn bản dài. ``${}`

Một số hàm xử lý xâu

Tên hàm	Mô tả
<code>charAt()</code>	Trả về ký tự ở chỉ mục đã cho
<code>concat()</code>	Trả về kết hợp của hai hoặc nhiều chuỗi được chỉ định
<code>indexOf()</code>	Trả về một chỉ mục xuất hiện đầu tiên của chuỗi con được chỉ định từ một chuỗi (-1 nếu không tìm thấy)
<code>replace()</code>	Thay thế chuỗi con phù hợp bằng một chuỗi con mới
<code>split()</code>	Chia chuỗi thành chuỗi con và trả về một mảng
<code>toUpperCase()</code>	Chuyển đổi tất cả các ký tự của chuỗi thành chữ hoa
<code>toLowerCase()</code>	Chuyển đổi tất cả các ký tự của chuỗi thành chữ thường
<code>charCodeAt()</code>	Trả về một số là giá trị đơn vị mã UTF-16 tại chỉ mục đã cho
<code>codePointAt()</code>	Trả về số nguyên không âm là giá trị điểm mã của điểm mã được mã hóa UTF-16 bắt đầu từ chỉ mục đã chỉ định
<code>includes()</code>	Kiểm tra xem một chuỗi có bao gồm một chuỗi khác không
<code>endsWith()</code>	Kiểm tra xem một chuỗi kết thúc bằng một chuỗi khác
<code>LastIndexOf()</code>	Trả về chỉ số xuất hiện lần cuối của giá trị trong chuỗi
<code>localeCompare()</code>	Kiểm tra xem một chuỗi đến trước, sau hay giống với chuỗi đã cho
<code>match()</code>	Khớp một biểu thức chính quy với chuỗi đã cho
<code>normalize()</code>	Trả về dạng chuẩn hóa Unicode của chuỗi đã cho.
<code>padEnd()</code>	Đệm phần cuối của chuỗi hiện tại bằng chuỗi đã cho
<code>padStart()</code>	Đệm phần đầu của chuỗi hiện tại bằng chuỗi đã cho

Tên hàm	Mô tả
repeat()	Trả về một chuỗi bao gồm các phần tử của đối tượng được lặp lại trong thời gian đã cho.
search()	Tìm kiếm sự trùng khớp giữa một biểu thức chính quy và một chuỗi
slice()	Trả về một phần của chuỗi
startsWith()	Kiểm tra xem một chuỗi bắt đầu bằng một chuỗi khác
substr()	Trả về một chuỗi bắt đầu tại vị trí đã chỉ định và của các ký tự đã cho
substring()	Trả về một chuỗi giữa hai chỉ mục đã cho
toLocaleLowerCase()	Trả về một chuỗi chữ thường trong khi tôn trọng miền địa phương hiện tại
toLocaleUpperCase()	Trả về một chuỗi chữ hoa trong khi tôn trọng miền địa phương hiện tại
trim()	Cắt khoảng trắng từ đầu và cuối chuỗi
trimLeft()	Cắt khoảng trắng từ phía bên trái của chuỗi
trimRight()	Cắt khoảng trắng từ phía bên phải của chuỗi

Thông tin chi tiết các em tra cứu trong địa chỉ sau:

https://www.tutorialspoint.com/typescript/typescript_strings.htm

2.6.2 Kiểu dữ liệu Arrays

Mảng là một loại dữ liệu đặc biệt có thể lưu trữ nhiều giá trị của các loại dữ liệu khác nhau theo cách sử dụng một cú pháp đặc biệt.

TypeScript hỗ trợ các mảng, tương tự như JavaScript. Có hai cách để khai báo một mảng:

Sử dụng dấu ngoặc vuông. Phương pháp này tương tự như cách bạn khai báo mảng trong JavaScript.

```
let fruits: string[] = ['Apple', 'Orange', 'Banana'];
```

Sử dụng kiểu mảng chung, Mảng <ElementType>.

```
let fruits: Array<string> = ['Apple', 'Orange', 'Banana'];
```

Cả hai phương pháp đều tạo ra cùng một đầu ra.

Tất nhiên, bạn luôn có thể khởi tạo một mảng như hiển thị bên dưới, nhưng bạn sẽ không nhận được lợi thế của hệ thống loại TypeScript.

```
let arr = [1, 3, 'Apple', 'Orange', 'Banana', true, false];
```

Mảng có thể chứa các phần tử của bất kỳ loại dữ liệu, số, chuỗi hoặc thậm chí các đối tượng.

Mảng có thể được khai báo và khởi tạo riêng.

Ví dụ: Khai báo và khởi tạo mảng

```
let fruits: Array<string>;  
fruits = ['Apple', 'Orange', 'Banana'];  
let ids: Array<number>;  
ids = [23, 34, 100, 124, 44];
```

Một mảng trong TypeScript có thể chứa các thành phần của các loại dữ liệu khác nhau bằng cách sử dụng cú pháp kiểu mảng chung, như được hiển thị bên dưới.

Ví dụ: Mảng nhiều loại

```
let values: (string | number)[] = ['Apple', 2, 'Orange', 3, 4, 'Banana'];  
// or  
let values: Array<string | number> = ['Apple', 2, 'Orange', 3, 4, 'Banana'];
```

Truy cập các phần tử mảng:

Các phần tử mảng có thể được truy cập bằng cách sử dụng chỉ mục của một phần tử, vd `ArrayName[index]`. Chỉ số mảng bắt đầu từ 0, vì vậy chỉ mục của phần tử thứ nhất bằng 0, chỉ mục của phần tử thứ hai là một và cứ thế.

Ví dụ: Các phần tử mảng truy cập

```
let fruits: string[] = ['Apple', 'Orange', 'Banana'];  
fruits[0]; // returns Apple  
fruits[1]; // returns Orange  
fruits[2]; // returns Banana  
fruits[3]; // returns undefined
```

Sử dụng vòng lặp for để truy cập các phần tử mảng như dưới đây.

Ví dụ: Access Array Elements bằng Loop

```
let fruits: string[] = ['Apple', 'Orange', 'Banana'];  
for(var index in fruits)  
{
```

```

    console.log(fruits[index]); // output: Apple Orange Banana
  }
  for(var i = 0; i < fruits.length; i++)
  {
    console.log(fruits[i]); // output: Apple Orange Banana
  }

```

Một số hàm thông dụng trên mảng

STT	Hàm và mô tả
1	concat() Trả về một mảng mới bao gồm mảng này được nối với (các) mảng và / hoặc giá trị khác.
2	every() Trả về true nếu mọi phần tử trong mảng này thỏa mãn chức năng kiểm tra được cung cấp.
3	filter() Tạo một mảng mới với tất cả các thành phần của mảng này mà hàm lọc được cung cấp trả về true. Ví dụ 1: <pre>function isBigEnough(element, index, array) { return (element >= 10); }</pre> <pre>var passed = [12, 5, 8, 130, 44].filter(isBigEnough); console.log("Test Value : " + passed);</pre> Kết quả: Test Value :12,130,44 Ví dụ 2: <pre>var passed = [12, 5, 8, 130, 44].filter(x => x>=10); console.log("Test Value : " + passed);</pre> Kết quả: Test Value :12,130,44
4	forEach()

	<p>Gọi một hàm cho mỗi phần tử trong mảng.</p> <p>Ví dụ 1:</p> <pre>let num = [7, 8, 9]; num.forEach(function (value) { console.log(value); });</pre> <p>Kết quả:</p> <pre>7 8 9</pre> <p>Ví dụ 2:</p> <pre>let result = ""; let myArray = [{name:"a"}, {name:""}, {name:"b"}, {name:"c"}]; myArray.forEach((myObject, index) => { if(myObject.name){ result +=myArray[index].name; } });</pre> <p>Kết quả: abc</p>
5	<p>indexOf()</p> <p>Trả về chỉ mục (ít nhất) đầu tiên của một phần tử trong mảng bằng với giá trị được chỉ định hoặc -1 nếu không tìm thấy.</p>
6.	<p>join()</p> <p>Nối tất cả các phần tử của một mảng thành một chuỗi.</p>
7	<p>lastIndexOf()</p> <p>Trả về chỉ mục (lớn nhất) cuối cùng của một phần tử trong mảng bằng với giá trị được chỉ định hoặc -1 nếu không tìm thấy.</p>
8	<p>map()</p> <p>Tạo một mảng mới với kết quả gọi một hàm được cung cấp trên mọi phần tử trong</p>

mảng này.

Ví dụ 1:

```
var numbers = [1, 4, 9];
var roots = numbers.map(Math.sqrt);
console.log("roots is : " + roots );
```

Kết quả: roots is : 1,2,3

Ví dụ 2:

```
const myUsers = [
  { name: 'shark', likes: 'ocean' },
  { name: 'turtle', likes: 'pond' },
  { name: 'otter', likes: 'fish biscuits' }
]
const usersByLikes = myUsers.map(item => {
  const container = {};

  container[item.name] = item.likes;
  container.age = item.name.length * 10;

  return container;
})
console.log(usersByLikes);
```

Kết quả:

```
[
  {shark: "ocean", age: 50},
  {turtle: "pond", age: 60},
  {otter: "fish biscuits", age: 50}
]
```

9

pop()

Loại bỏ phần tử cuối cùng khỏi một mảng và trả về phần tử đó.

10

push()

	Thêm một hoặc nhiều phần tử vào cuối một mảng và trả về độ dài mới của mảng.
11	<p>reduce()</p> <p>Áp dụng đồng thời một hàm đối với hai giá trị của mảng (từ trái sang phải) để giảm nó thành một giá trị.</p> <p>Ví dụ 1:</p> <pre>var total = [0, 1, 2, 3].reduce(function(a, b){ return a + b; }); console.log("total is : " + total);</pre> <p>Kết quả: <i>total is :6</i></p> <p>Ví dụ 2:</p> <pre>let array=[5,4,19,2,7]; function findMax(acc,val) { if(val>acc){ acc=val; } } let biggest=array.reduce(findMax); // 19</pre>
12	<p>reduceRight()</p> <p>Áp dụng đồng thời một hàm đối với hai giá trị của mảng (từ phải sang trái) để giảm nó thành một giá trị.</p>
13	<p>reverse()</p> <p>Đảo ngược thứ tự các phần tử của một mảng - cái đầu tiên trở thành cái cuối cùng và cái cuối cùng trở thành cái đầu tiên.</p>
14	<p>shift()</p> <p>Loại bỏ phần tử đầu tiên khỏi một mảng và trả về phần tử đó.</p>
15	<p>slice()</p> <p>Trích xuất một phần của một mảng và trả về một mảng mới.</p>
16	some()

	Trả về true nếu có ít nhất một phần tử trong mảng này thỏa mãn chức năng kiểm tra được cung cấp.
17	sort() Sắp xếp các phần tử của một mảng.
18	splice() Thêm và / hoặc loại bỏ các phần tử từ một mảng.
19	toString() Trả về một chuỗi đại diện cho mảng và các phần tử của nó.
20	unshift() Thêm một hoặc nhiều phần tử vào phía trước của một mảng và trả về độ dài mới của mảng.

Thông tin chi tiết các em tra cứu trong địa chỉ sau:

https://www.tutorialspoint.com/typescript/typescript_arrays.htm

2.7 Lập trình hướng đối tượng

Trong các ngôn ngữ lập trình hướng đối tượng như Java và C #, các lớp là các thực thể cơ bản được sử dụng để tạo các thành phần có thể sử dụng lại. Các chức năng được truyền lại cho các lớp và các đối tượng được tạo từ các lớp. Tuy nhiên, cho đến ECMAScript 6 (còn được gọi là ECMAScript 2015), đây không phải là trường hợp của JavaScript. JavaScript chủ yếu là một ngôn ngữ lập trình chức năng trong đó tính kế thừa dựa trên nguyên mẫu. Các chức năng được sử dụng để xây dựng các thành phần có thể tái sử dụng. Trong ECMAScript 6, cách tiếp cận dựa trên lớp hướng đối tượng đã được giới thiệu. TypeScript đã giới thiệu các lớp để tận dụng lợi ích của các kỹ thuật hướng đối tượng như đóng gói và trừu tượng hóa. Lớp trong TypeScript được biên dịch thành các hàm JavaScript đơn giản bởi trình biên dịch TypeScript để hoạt động trên các nền tảng và trình duyệt.

Một lớp có thể bao gồm:

- Constructor
- Tính chất
- Phương thức

Sau đây là một ví dụ về một lớp trong TypeScript:

Class

Ví dụ:

```
class Employee {
    empCode: number;
    empName: string;
    constructor(code: number, name: string) {
        this.empName = name;
        this.empCode = code;
    }
    getSalary(): number {
        return 10000;
    }
}
```

Trình biên dịch TypeScript sẽ chuyển đổi lớp trên thành mã JavaScript sau bằng cách sử dụng bao đóng :

```
var Employee = /** @class */ (function () {
    function Employee(name, code) {
        this.empName = name;
        this.empCode = code;
    }
    Employee.prototype.getSalary = function () {
        return 10000;
    };
    return Employee;
})();
```

Constructor

Hàm tạo là một loại phương thức đặc biệt được gọi khi tạo đối tượng. Trong TypeScript, phương thức constructor luôn được định nghĩa với tên "constructor".

Ví dụ: Employee

```
class Employee {
    empCode: number;
```

```
empName: string;  
constructor(empcode: number, name: string ) {  
    this.empCode = empcode;  
    this.name = name;  
}  
}
```

Trong ví dụ trên, Employee lớp bao gồm một hàm tạo với các tham số empcode và name. Trong hàm tạo, các thành viên của lớp có thể được truy cập bằng this từ khóa this.empCode hoặc this.name.

Không cần thiết cho một lớp để có một hàm tạo.

Ví dụ: Lớp không có Trình xây dựng

```
class Employee {  
    empCode: number;  
    empName: string;  
}
```

Tạo một đối tượng của lớp

Một đối tượng của lớp có thể được tạo bằng cách sử dụng từ khóa mới .

Ví dụ: Tạo một đối tượng

```
class Employee {  
    empCode: number;  
    empName: string;  
}  
  
let emp = new Employee();
```

Ở đây, chúng ta tạo một đối tượng được gọi là emp kiểu Employee sử dụng let emp = new Employee();. Lớp trên không bao gồm bất kỳ hàm tạo tham số nào, vì vậy chúng ta không thể truyền các giá trị trong khi tạo một đối tượng. Nếu lớp bao gồm một hàm tạo được tham số hóa, thì chúng ta có thể truyền các giá trị trong khi tạo đối tượng.

```
class Employee {  
    empCode: number;  
    empName: string;  
    constructor(empcode: number, name: string ) {  
        this.empCode = empcode;
```

```

        this.name = name;
    }
}

let emp = new Employee(100, "Steve");

```

Trong ví dụ trên, chúng ta truyền các giá trị cho đối tượng để khởi tạo các biến thành viên. Khi chúng ta khởi tạo một đối tượng mới, hàm tạo của lớp được gọi với các giá trị được truyền và các biến thành viên empCode và empName được khởi tạo với các giá trị này.

Kế thừa

Giống như các ngôn ngữ hướng đối tượng như Java và C #, các lớp TypeScript có thể được mở rộng để tạo các lớp mới có tính kế thừa, sử dụng từ khóa extends.

Ví dụ: Kế thừa

```

class Person {
    name: string;
    constructor(name: string) {
        this.name = name;
    }
}

class Employee extends Person {
    empCode: number;
    constructor(empcode: number, name:string) {
        super(name);
        this.empCode = empcode;
    }
    displayName():void {
        console.log("Name = " + this.name + ", Employee Code = " +
this.empCode);
    }
}

let emp = new Employee(100, "Bill");
emp.displayName(); // Name = Bill, Employee Code = 100

```

Trong ví dụ trên, Employee lớp mở rộng từ lớp Person bằng cách sử dụng từ extends. Điều này có nghĩa là lớp Employee hiện bao gồm tất cả các thành viên của lớp Person.

Hàm tạo của lớp Employee khởi tạo các thành viên của chính nó cũng như các thuộc tính của lớp cha bằng cách sử dụng một từ khóa đặc biệt ' *super* '. Các từ khóa *super* được sử dụng để gọi constructor lớp cha.

Chú ý: Chúng ta phải gọi phương thức *super* () trước khi gán giá trị cho các thuộc tính trong hàm tạo của lớp dẫn xuất.

Interface

Một lớp có thể thực hiện một hoặc nhiều giao diện.

Ví dụ: Giao diện triển khai

```
interface IPerson {
    name: string;
    display():void;
}

interface IEmployee {
    empCode: number;
}

class Employee implements IPerson, IEmployee {
    empCode: number;
    name: string;
    constructor(empcode: number, name:string) {
        this.empCode = empcode;
        this.name = name;
    }
    display(): void {
        console.log("Name = " + this.name + ", Employee Code = " +
this.empCode);
    }
}

let per:IPerson = new Employee(100, "Bill");
per.display(); // Name = Bill, Employee Code = 100
```



```
let emp:IEmployee = new Employee(100, "Bill");
emp.display(); //Compiler Error: Property 'display' does not exist on type
'IEmployee'
```

Giao diện mở rộng lớp

Một giao diện cũng có thể mở rộng một lớp để thể hiện một loại.

Ví dụ: Giao diện mở rộng lớp

```
class Person {
    name: string;
}
interface IEmployee extends Person {
    empCode: number;
}
let emp: IEmployee = { empCode : 1, name:"James Bond" }
```

Trong ví dụ trên, Iemployee là một giao diện mở rộng lớp Person. Vì vậy, chúng ta có thể khai báo một biến loại Iemployee có hai thuộc tính. Vì vậy, bây giờ, chúng ta phải khai báo và khởi tạo các giá trị cùng một lúc.

Phương pháp ghi đè

Khi một lớp con định nghĩa việc thực hiện phương thức riêng của nó từ lớp cha, nó được gọi là phương thức ghi đè.

Ví dụ: Phương thức ghi đè

```
class Car {
    name: string;
    constructor(name: string) {
        this.name = name;
    }
    run(speed:number = 0) {
        console.log("A " + this.name + " is moving at " + speed + " mph!");
    }
}
class Mercedes extends Car {
    constructor(name: string) {
        super(name);
    }
}
```

```
}  
run(speed = 150) {  
  console.log('A Mercedes started')  
  super.run(speed);  
}  
}  
  
class Honda extends Car {  
  constructor(name: string) {  
    super(name);  
  }  
  run(speed = 100) {  
    console.log('A Honda started')  
    super.run(speed);  
  }  
}  
  
let mercObj = new Mercedes("Mercedes-Benz GLA");  
let hondaObj = new Honda("Honda City")  
mercObj.run(); // A Mercedes started A Mercedes-Benz GLA is moving at 150  
mph!  
hondaObj.run(); // A Honda started A Honda City is moving at 100 mph!
```

Trong ví dụ trên, chúng ta có một lớp Car với thuộc tính name. Hàm tạo cho lớp này khởi tạo các biến thành viên. Lớp này cũng có một phương thức display() với tốc độ đổi số được khởi tạo là 0.

Sau đó chúng ta tạo hai lớp Mercedes và Honda, mở rộng từ lớp cha Car. Mỗi lớp con mở rộng các thuộc tính của lớp cha. Hàm tạo cho mỗi lớp gọi siêu xây dựng để khởi tạo các thuộc tính của lớp cha. Mỗi lớp cũng định nghĩa một phương thức run() trong thông điệp riêng của mình ngoài việc gọi phương thức siêu lớp cho run().

Vì mỗi lớp con có cách thực hiện riêng của phương thức run(), nên nó được gọi là phương thức ghi đè, tức là các lớp con có một phương thức cùng tên với phương thức của lớp cha.

Khi chúng ta tạo các đối tượng của lớp con và gọi run() phương thức trên đối tượng này, nó sẽ gọi phương thức được ghi đè của chính nó run() và không phải là phương thức của lớp cha.

Bài 3: Component

3.1 Component là gì?

Components là một khối code trong app Angular. Nó là sự kết hợp của bộ template html (bộ khung html) và nhúng kèm code TypeScript (hoặc Javascript). Các components là độc lập với nhau và độc lập với hệ thống. Nó có thể được cài vào hoặc tháo ra khỏi hệ thống dễ dàng. Một component có thể hiểu như một control trên màn hình hiển thị, gồm giao diện html và code logic xử lý sự kiện đi kèm control đó. Một component cũng có thể to lớn như là cả 1 màn hình chứa nhiều control hoặc một nhóm nhiều màn hình. Tức là là một component cũng có thể chứa và gọi được nhiều component khác nối vào. Như vậy Angular rất linh hoạt trong việc chia nhỏ code ra các component.

Trong Angular chúng ta khai báo một Component với cấu trúc như sau:

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-hello-world',
  template: `<h1>Hello Angular world</h1>`
})
export class HelloWorld {
}
```

Như chúng ta thấy từ khóa @Component sẽ giúp định nghĩa ra một bộ khung html cho nó. Và bên dưới là một class HelloWorld dùng để viết code logic. Trong định nghĩa bộ khung html, chúng ta có một số thuộc tính cần chú ý sau đây:

- **selector** : Là tên được đặt để gọi một component trong code html. Ở ví dụ vừa rồi, từ khóa app-hello-world được đặt tên cho component này. Khi cần gọi component này ra ở màn hình html cha, ta sẽ gọi bằng html tag <app-hello-world></app-hello-world>. Gọi như vậy thì component con sẽ được render ra component cha.
- **template** : Là tự định nghĩa khung html cho component dạng string ở trong file này luôn. Ví dụ ở trên chỉ định nghĩa một thẻ html h1 đơn giản. Cách này chỉ dùng cho component đơn giản.
- **templateUrl** : Là đường dẫn url tới file html bên ngoài để load file đó vào làm khung html cho component này. Đây là cách code hay được dùng vì cho phép

tách riêng khung html ra khỏi code logic, người làm design sẽ sửa file html riêng, độc lập với người làm code.

- **styles** : Là viết style css luôn vào file component này. Cách này chỉ dùng cho component đơn giản.
- **styleUrls** : Là đường dẫn url đến file style css độc lập cho component này. Cách này khuyên dùng vì file css nên để dành riêng cho người designer đung vào.

3.2 Thành phần cha AppComponent

Trong bài trước, khi tạo project, chúng ta đã thấy các file được tạo ra trong thư mục src/app:

```
app.component.css
app.component.html
app.component.spec.ts
app.component.ts
app.module.ts
```

Nếu bạn mở file app.module.ts ra, bạn sẽ thấy nội dung như sau:

app.module.ts

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { AppComponent } from './app.component';
@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Trong đó:

@NgModule: để định nghĩa Class AppModule là một Angular Module.

@NgModule: cũng sẽ có những metadata object, cái mà sẽ nói cho Angular biết cách biên dịch và khởi chạy ứng dụng của chúng ta.

Imports: import BrowserModule module, module này cần cho mọi ứng dụng chạy trên trình duyệt.

Declarations: Khai báo những component sẽ được sử dụng trong module này.

Bootstrap: Component gốc được Angular tạo ra và chèn vào trong index.html.

app.component.ts

```
import { Component } from '@angular/core';  
@Component({  
  selector: 'app-root',  
  templateUrl: './app.component.html',  
  styleUrls: ['./app.component.css']  
})  
export class AppComponent {  
  title = 'Ứng dụng Smart note';  
}
```

AppComponent là thành phần cha của ứng dụng. Tất cả các thành phần mới tạo ra sau này đều là thành phần con của AppComponent.

3.3 Tạo một Component

Để tạo component mới, ta sử dụng lệnh Angular CLI:

```
ng g component <ten-component>
```

Ví dụ, để tạo component có tên là new-cmp, ta chạy lệnh `ng g component new-cmp`

Sau khi chạy, màn hình hiển thị quá trình tạo component. Bạn đợi chừng 5 - 10s thì...

bùm, component new-cmp được tạo ra như hình dưới

```
D:\Angular\HelloWorld>ng g component new-cmp  
CREATE src/app/new-cmp/new-cmp.component.html (26 bytes)  
CREATE src/app/new-cmp/new-cmp.component.spec.ts (629 bytes)  
CREATE src/app/new-cmp/new-cmp.component.ts (272 bytes)  
CREATE src/app/new-cmp/new-cmp.component.css (0 bytes)  
UPDATE src/app/app.module.ts (477 bytes)
```

Vào kiểm tra cấu trúc thư mục thì ta thấy thư mục new-cmp được tạo ra trong thư mục src/app/. Trong thư mục new-cmp có 4 file được tạo ra:

new-cmp.component.css – file css của component.

new-cmp.component.html – file html của component.

new-cmp.component.spec.ts – file sử dụng cho unit test.

new-cmp.component.ts – file chứa code xử lý logic

Ngoài việc tạo ra mới 4 file này, angular CLI cũng giúp chúng ta khai báo new component này cho project trong file app.module.ts

app.module.ts

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { AppComponent } from './app.component';
import { NewCmpComponent } from './new-cmp/new-cmp.component';
@NgModule({
  declarations: [
    AppComponent,
    NewCmpComponent // Dòng này mới được thêm vào
  ],
  imports: [
    BrowserModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

3.4 Cấu trúc chung của một Component

new-cmp.component.ts

```
// Khai báo import angular/core
import { Component, OnInit } from '@angular/core';
@Component({
  // Khai báo selector cho component new-cmp
```

```

    selector: 'app-new-cmp',
    // Khai báo file html mà component new-cmp "đại diện"
    templateUrl: './new-cmp.component.html',
    // Khai báo file style sử dụng
    styleUrls: ['./new-cmp.component.css'] })
export class NewCmpComponent implements OnInit {
    constructor() { }
    ngOnInit() {}
}

```

Ở đây khai báo 1 class gọi là NewCmpComponent. Class này implements class OnInit. Class OnInit có constructor là ngOnInit(), nên trong NewCmpComponent cũng sẽ override lại constructor ngOnInit() này. Constructor ngOnInit() sẽ được gọi mặc định khi class được gọi đến.

Ở bài trước, ta đã tìm hiểu được luồng hoạt động của AppComponent: file main.ts được chạy đầu tiên. Trong file main.ts khai báo AppModule. AppModule khai báo sử dụng AppComponent. Trong App Component có selector <app-root>, <app-root> được gọi trong file index.html và cuối cùng file index.html được hiển thị lên trình duyệt.

Cách hoạt động của component khác cũng tương tự như App Component. Trong ví dụ trên, NewCmpComponent khai báo selector là <app-new-cmp>. Như vậy thẻ <app-new-cmp> sẽ "đại diện" cho NewCmpComponent, khai báo <app-new-cmp> ở đâu thì nội dung file html của NewCmpComponent sẽ được hiển thị ở chỗ đó, giống như khi khai báo <app-root></app-root> ở đâu thì nội dung file app.component.html được hiển thị ở đó.

3.5 Sử dụng component

Để sử dụng component, bạn cần đặt selector của component ở nơi cần hiển thị.

Lý thuyết nhiều hơi khó hiểu nên chúng ta hãy cùng làm 1 ví dụ về sử dụng component cho dễ hình dung nhé.

Giả sử tớ muốn sửa trang chủ của ứng dụng smart-note như hình dưới



Nhìn vào đây, ta có phân tích:

- NewCmpComponent chứa danh sách các công việc cần làm => lưu danh sách công việc cần làm trong NewComponent class, chạy vòng for ngoài file new-cmp.component.html trong cặp thẻ
- Gọi đến selector của NewCmpComponent trong app.component.html.

Đầu tiên, trong file new-cmp.component.ts, chúng ta thêm khai báo mảng dữ liệu:

new-cmp.component.ts

```
import { Component, OnInit } from '@angular/core';
@Component({
  selector: 'app-new-cmp',
  templateUrl: './new-cmp.component.html',
  styleUrls: ['./new-cmp.component.css']
})
export class NewCmpComponent implements OnInit {
  todo = ["Học TypeScript", "Học Angular", "Học HTML5"]; // Khai báo mảng
  dữ liệu
```



```

constructor() { }

ngOnInit() {

}

}

```

Sau đó, bên trong file `new-cmp.component.html`, ta chạy vòng `for` để hiển thị dữ liệu bên trong `component` ra

new-cmp.component.html

<p>Danh sách các công việc cần làm</p>

<li *ngFor="let cv of todo">{{cv}}

Component new-cmp có selector là app-new-cmp, do vậy, để hiển thị dữ liệu trong component ra trang chủ, ta chỉ cần gọi đến selector của component này trong component cha. Ta thêm selector bên trong file app.component.html

Để hiển thị dữ liệu ra màn hình thì trong file template (tức là file html khai báo trong Component) ta sẽ sử dụng cú pháp `{{ }}` hay còn gọi là phương thức nội suy Interpolation.

app.component.html

<div style="text-align:center">
 <h1>
 {{app}}
 </h1>

</div>
<!-- Đây là selector của component NewCmp -->
<app-new-cmp></app-new-cmp>

```

Ứng dụng lúc này của chúng ta đã "lên hình"



## Ứng dụng Smart note



Danh sách các công việc cần làm

- Học TypeScript
- Học Angular 4
- Học HTML5

Tiếp theo, chúng ta sẽ thêm css cho ứng dụng đẹp hơn. Ta thêm các class css: text-medium, text-large để thay đổi font-size, text-blue và text-red để thay đổi màu chữ. Mỗi component có 1 file css riêng để không ảnh hưởng đến các component khác. Ở đây, chúng ta sẽ thêm các class trong file new-cmp.component.css

### new-cmp.component.css

```

text-medium{
 font-size: 1.5rem;
}
.text-large{
 font-size: 1.9rem;
}

```

```

}

.text-red{
 color:red;
}

.text-blue{
 color:blue;
}

```

Sau đó thêm các class cho các thẻ như sau, ta sẽ thu được kết quả là ứng dụng như ban đầu đặt ra.

#### new-cmp.component.html

```

<p class="text-large text-red">Danh sách các công việc cần làm</p>
<ul class="text-medium">
 <li *ngFor="let cv of todo" class="text-blue">{{cv}}


```

#### Kết quả hiển thị

← → ↻ localhost:4200 ☆ ⚙️ 👤 🔄

#### Ứng dụng Smart note



#### Danh sách các công việc cần làm

- Học TypeScript
- Học Angular
- Học HTML5

### 3.6 Nhúng bootstrap, css, js vào project

Để nhúng bootstrap, css, js vào ứng dụng angular chúng ta có hai cách

Cách 1: Nhúng vào file index.html

Cách 2: Chèn vào file angular.json

```

"styles": [
 "src/assets/css/bootstrap.min.css",
 "src/assets/css/bootstro.min.css",

```

```

 "src/assets/font-awesome/5.1.1/css/all.min.css",
 "src/assets/css/custom.css",
 "src/styles.css"
],
 "scripts": [
 "node_modules/jquery/dist/jquery.min.js",
 "node_modules/tinymce/tinymce.min.js",
 "node_modules/moment/min/moment.min.js",
 "src/assets/js/lz-string.min.js",
 "src/assets/js/wizard.min.js",
 "src/assets/js/customizes.js",
 "src/assets/jspdf/jspdf.min.js"
]
}

```

### 3.7 Module

Module có thể hiểu là nơi gom góp tất cả các thành phần của 1 ứng dụng (component, directive, pipe, service) thành một thể thống nhất. Ví như module là 1 website thì component, directive, pipe, service giống như các phần header, navbar, menu, footer vậy.

#### Import NgModule

Để định nghĩa 1 module, bạn sử dụng NgModule.

Khi tạo project bằng Angular CLI, 1 module mặc định được tạo ra, đó là AppModule:

#### app.module.ts

```

import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { AppComponent } from './app.component';

@NgModule({
 declarations: [
 AppComponent
],
 imports: [
 BrowserModule
],

```

```
providers: [],
bootstrap: [AppComponent]
})

export class AppModule { }
```

NgModule được import như sau

#### **app.module.ts**

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

#### **Cấu trúc NgModule**

NgModule được khai báo thông qua 1 object có 4 thuộc tính: declarations, imports, providers, bootstrap.

```
@NgModule({
 declarations: [
 AppComponent
],
 imports: [
 BrowserModule
],
 providers: [],
 bootstrap: [AppComponent]
})
```

#### **Declaration**

Declaration là một mảng chứa các khai báo. Trong ví dụ trước, khi ta tạo thêm component thì declaration sẽ có thêm khai báo của NewCmpComponent như sau

#### **declaration**

```
declarations: [
 AppComponent,
 NewCmpComponent
]
```

#### **Import**

Import là một mảng chứa các module sử dụng trong ứng dụng hoặc các module sử dụng trong các component. Mặc định sẽ có `BrowserModule` được import.

Để import 1 module, chúng ta cần 2 bước: khai báo import module và khai báo trong mảng imports. Ví dụ chúng ta cần import module form cho ứng dụng, trước tiên ta cần khai báo module này được import từ `@angular/forms`

#### **import @angular/forms**

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

Sau đó, chúng ta thêm khai báo trong mảng import

#### **Khai báo FormsModule**

```
imports: [
 BrowserModule,
 FormsModule // Module form mới được thêm
]
```

#### **Providers**

Provider chứa các service mà chúng ta tạo ra

#### **Bootstrap**

Chứa component chính thực thi chương trình

## Bài 4: Template và Data Binding

Component cần một view để hiển thị. Template định nghĩa view. Template chỉ là một tập con của HTML, nó chỉ cho Angular biết làm sao để hiển thị view.

Ngôn ngữ chính của template là HTML, nhưng không phải các phân tử (hay các thẻ) đều hợp lệ với Angular, điển hình là thẻ `<script>`, với Angular thì `<script>` sẽ bị bỏ qua, không được biên dịch vì lý do bảo mật.

Chúng ta cũng có thể tự tạo ra các thẻ cho riêng mình thông qua component như trong các bài trước đã làm.

Data binding là cơ chế lấy dữ liệu từ trong component và đưa ra view

### 4.1 String interpolation

Để nhúng một biểu thức hoặc một biến từ component vào template chúng ta dùng phép nội suy (String interpolation) bằng cách đặt biến hoặc biểu thức trong dấu ngoặc kép `{{ ... }}`

Ví dụ:

```
//component.html
<p>{{ name }}</p>

// component.ts
@Component({
 templateUrl: 'component.html',
 selector: 'app-component',
})
export class Component {
 name = 'Peter';
}
```

### 4.2 Property Binding

Cơ chế binding này giúp bạn thiết lập các thuộc tính cho element trong view. Cập nhật giá trị của một thuộc tính trong view và binding nó đến một element. Cú pháp: `[]`. Sau đây là một ví dụ về thiết lập thuộc tính disabled qua property binding cho một button:

```
// component.htm
<button [disabled]="buttonDisabled"></button>

// component.ts
```

```
@Component({
 templateUrl: 'component.html',
 selector: 'app-component',
})
export class Component {
 buttonDisabled = true;
}
```

### 4.3 Event Binding

Khi người dùng tương tác với ứng dụng bằng các cách khác nhau: dùng bàn phím, click chuột, di chuyển chuột thì đều sinh ra một sự kiện. Các sự kiện này cần được kiểm soát (handle) để thực hiện hành động theo ý muốn của lập trình viên.

Trong bài này, chúng ta sẽ tìm hiểu về event binding (hay còn gọi là gán, kiểm soát các sự kiện) trong Angular.

#### Cách sử dụng

*(event\_name) = "function\_name(\$event)"*

Trong đó:

*event\_name: tên event (click, mouseover, mouseenter, ...)*

*function\_name: tên function trong file component.ts*

Ví dụ, bạn tạo một project mới từ quickstart, sau đó sửa trong file app.component.ts như sau:

#### app.component.ts

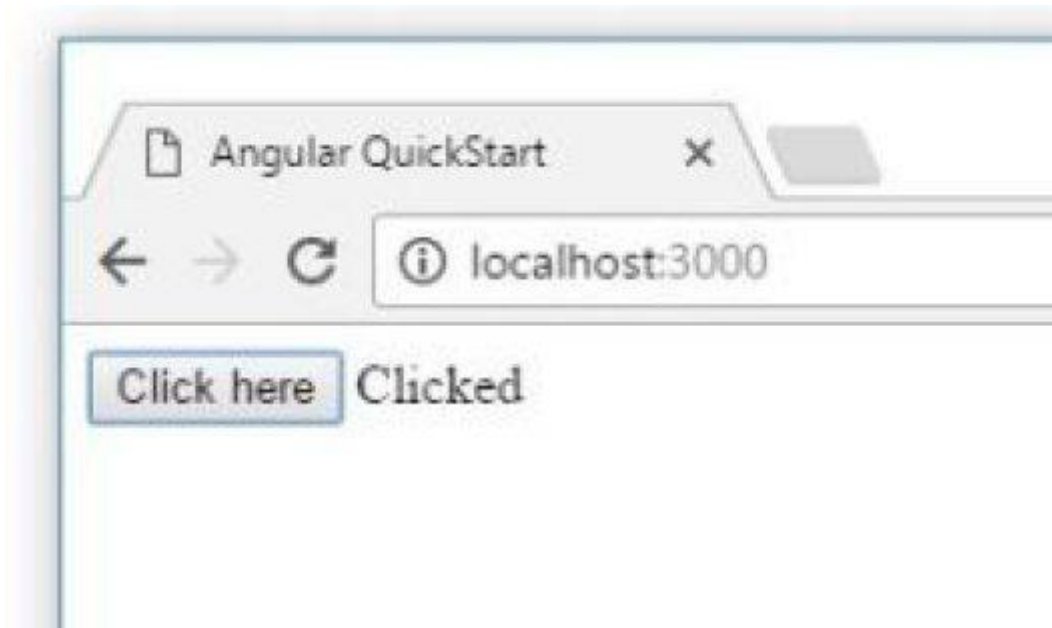
```
import { Component } from '@angular/core';

@Component({
 selector: 'my-app',
 template: `
 <button (click)="onClick()">Click here</button>
 {{ message }}
 `,
})
export class AppComponent {
 message = "";
```



```
onClick() {
 this.message = "Clicked";
}
}
```

Rất đơn giản, chúng ta khai báo trong lớp AppComponent một thuộc tính là message, và khi click thì phương thức onClick() sẽ được gọi, trong đó chúng ta gán thuộc tính message giá trị là “Clicked”. Giá trị của thuộc tính này sẽ được hiển thị trên template.



### Lấy dữ liệu từ sự kiện từ đối tượng \$event

Có một số sự kiện sẽ mang theo cả dữ liệu, chẳng hạn như click chuột thì có thể lấy được tọa độ chuột, bấm phím thì lấy được kí tự phím vừa bấm... v.v Chúng ta có thể lấy được các dữ liệu đó.

Ví dụ chúng ta sửa lại lớp AppComponent như sau:

#### app.component.ts

```
import { Component } from '@angular/core';
@Component({
 selector: 'my-app',
 template: `
 Type here: <input (keyup)="onKey($event)">
 <p>You typed: {{values}}</p>
 `
})
```

```
export class AppComponent {
 values = '';
 onKey(event: any) {
 this.values = event.target.value;
 }
}
```

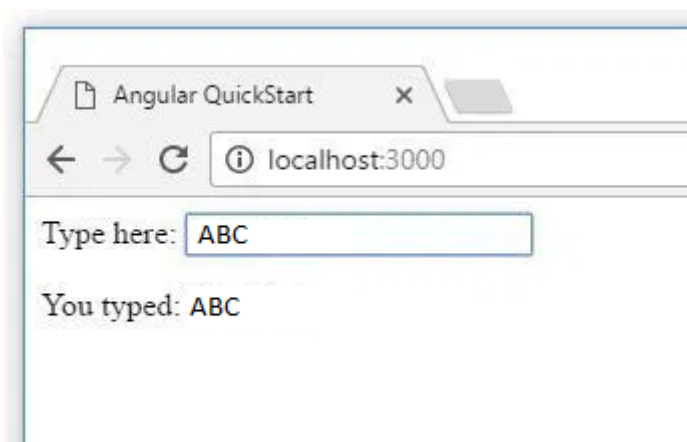
Ở đây chúng ta khai báo template là một thẻ `<input>`, với sự kiện `keyup` xử lý bằng phương thức `onKey()`. Dữ liệu sẽ được truyền vào tham số `$event`.

Khi người dùng nhấn và thả một phím, sự kiện `keyup` sẽ xảy ra, Angular sẽ truyền đối tượng sự kiện DOM là biến `$event` vào làm tham số của phương thức `onKey()`.

Tùy vào sự kiện là gì mà đối tượng `$event` sẽ có những thông tin khác nhau.

Tất cả các đối tượng sự kiện DOM đều có một thuộc tính là `target`, thuộc tính này tham chiếu đến thẻ đã phát sinh ra sự kiện đó, ở đây là `<input>`, và chúng ta có thể lấy thuộc tính `value` của thẻ này để lấy nội dung của thẻ.

Trong phương thức `onKey()` ở đây chúng ta gán giá trị cho biến `values` là giá trị trong `event.target.value`.



### Lấy dữ liệu trong template

Thay vì khai báo phương thức cho sự kiện, chúng ta có thể tham chiếu đến dữ liệu của sự kiện đó ngay trong template như sau:

#### **app.component.ts**

```
import { Component } from '@angular/core';
@Component({
 selector: 'my-app',
 template: `
```

```

 Type here: <input #box>
 <p>You typed: {{box.value}}</p>
 ,
 })
 export class AppComponent {

```

Để làm việc này thì chúng ta khai báo tên biến cho template, bằng cách ghi tên kèm với dấu #, ở đây chúng ta khai báo là #box. Ở phần bắt sự kiện chúng ta điền vào là 0, và chúng ta có thể dùng tên biến template đó để lấy giá trị của chính thẻ đó mà không cần phải dùng tới phương thức.

### Lọc sự kiện

Đôi khi chúng ta chỉ cần muốn bắt một giá trị cụ thể, chẳng hạn như phím Enter, chúng ta có thể bắt giá trị đó thông qua thuộc tính \$event.keyCode. Ví dụ:

#### app.component.ts

```

import { Component } from '@angular/core';
@Component({
 selector: 'my-app',
 template: `
 Type here: <input #box (keyup.enter)="onEnter(box.value)">
 <p>You typed: {{ value }}</p>
 ,
 })
 export class AppComponent {
 value = '';
 onEnter(value: string) {
 this.value = value;
 }
 }
}

```

Ở đây sự kiện sẽ xảy ra khi người dùng gõ phím Enter, chúng ta truyền vào giá trị của #box.value rồi gán giá trị đó vào biến value để hiển thị.

### 4.4 Two-way binding

Two-way Binding trong Angular chính là các thức tự động đồng bộ (synchronization) dữ liệu giữa Component và View trong Angular. Như các bài học lần trước các bạn đã

học cách đổ dữ liệu từ bên component ra view thì với việc sử dụng two-way Binding trong angular thì nếu dữ liệu ở view thay đổi thì bên trong component cũng sẽ thay đổi cùng một lúc và ngược lại nếu component thay đổi thì view cũng sẽ thay đổi theo. Nói tóm lại nó sẽ binding dữ liệu hai chiều.

Angular có một directive để áp dụng two-way binding đó là ngModel. Chúng ta có thể bao ngModel trong cặp dấu [], nó sẽ thực hiện đồng bộ dữ liệu từ Component với DOM và ngược lại.

#### **app.component.html**

```
<div>
 <input type="text"
 (keyup.enter)="updateMesssages()"
 [(ngModel)]="message" >
 <button (click)="updateMesssages()">Button...</button>
</div>
```

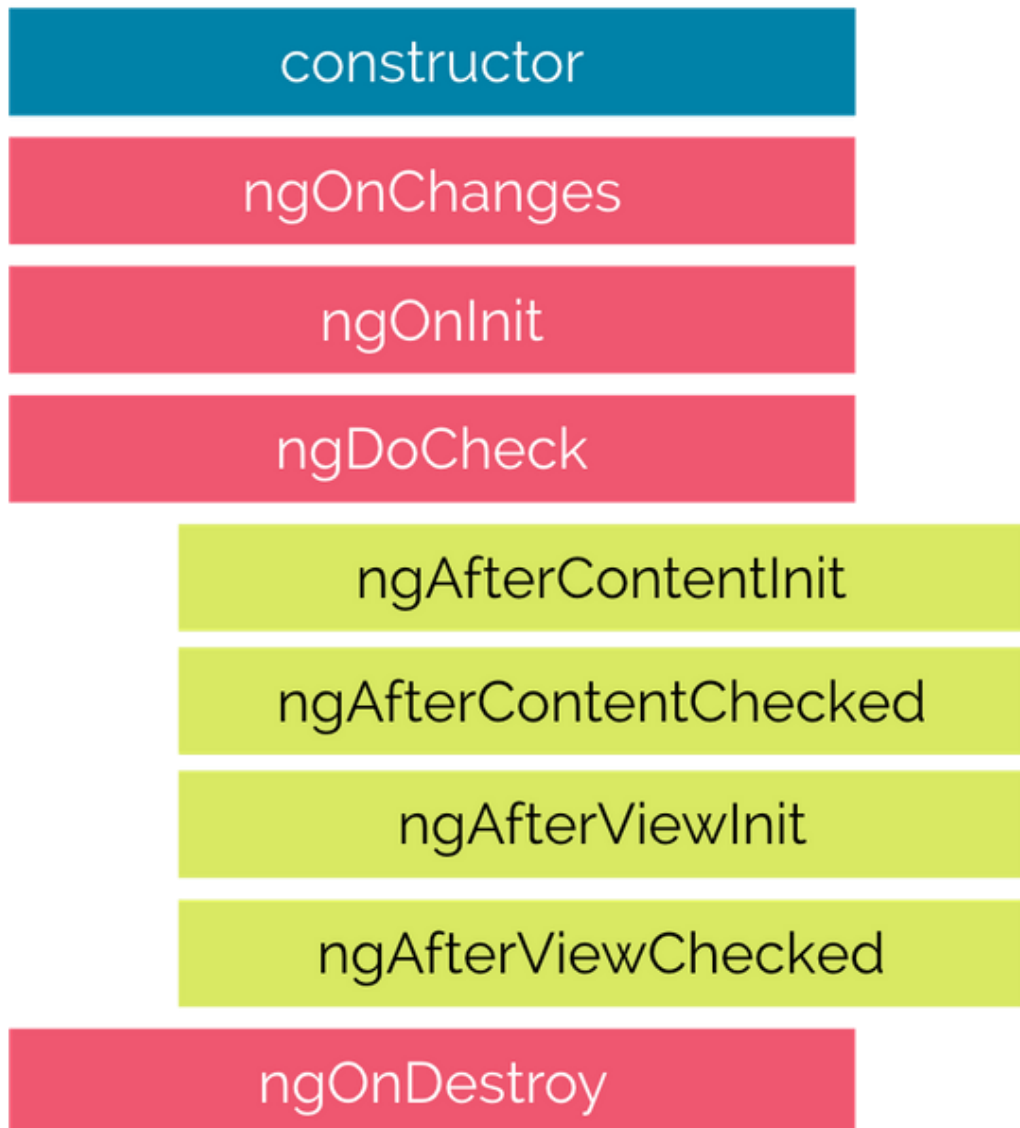
#### **app.component.ts**

```
import { Component } from '@angular/core';
@Component({
 selector: 'app-root',
 templateUrl: './app.component.html',
 styleUrls: ['./app.component.scss']
})
export class AppComponent {
 messages: string[] = [];
 message: string = "";
 updateMesssages() {
 this.messages.push(this.message);
 this.message = "";
 }
}
```

### **4.5 Vòng đời component**

Vòng đời của một component Angular hỗ trợ các Interface để tạo nên sự tuần tự trong quá trình hoạt động. Một vòng đời của Component sẽ có thứ tự như sau:

OnChanges, OnInit, DoCheck AfterContentInit, AfterContentChecked, AfterViewInit, AfterViewChecked, OnDestroy.



Để hiểu rõ hơn vòng đời của Component vào trang chủ của Angular để tìm hiểu chi tiết

#### **4.6 Giao tiếp giữa các component bằng @Input và @Output**

Ở component con muốn nhận dữ liệu từ component cha, thì ở component con sử dụng decorator @Input, cách sử dụng như sau:

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

@Component({
 selector: 'child-component',
```

```

 template: `
 <h3>Hello, {{hello}}</h3>
 `
 })
 export class ChildComponent {
 @Input() hello: string;
 };

```

Khi đó, ở component cha ta có thể truyền dữ liệu cho con theo cách bình thường:

```

import { Component } from '@angular/core';
@Component({
 selector: 'parent-component',
 template: `
 <h2>Parent component</h2>
 <child-component [hello]="truongluu"></child-component>
 `
})
export class ParentComponent {
}

```

**Lưu ý:** khi sử dụng decorator `@Input`, ta có thêm lựa chọn là thiết lập tham số đầu vào, nó là alias của thuộc tính truyền vào và biến bên trong component con. Cách sử dụng như sau :

```
@Input('message') hello: string;
```

và lúc này dữ liệu được truyền vào từ cha sẽ viết lại như sau:

```

@Component({
 selector: 'parent-component',
 template: `
 <h2>Parent component</h2>
 <child-component [message]="truongluu"></child-component>
 `
})

```

Ngược lại, từ component con muốn truyền dữ liệu cho cha, thì ta có thêm khái niệm `EventEmitter` (Người phát sự kiện), là khi ở con có một sự kiện gì đó làm thay đổi dữ

liệu, mà ở cha muốn nhận dữ liệu đó thì con sẽ đẩy dữ liệu này cho cha thông qua sử dụng `@Output` và `EventEmitter` ở con (hay còn gọi là tùy chỉnh sự kiện). Cách sử dụng như sau:

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

@Component({
 selector: 'child-component',
 template: `
 <h3>Child component, Hello: {{hello}}</h3>
 <button (click)="changeMessage()">Change message to "mono"</button>
 `
})
export class ChildComponent {
 @Input() hello: string;
 @Output() messageChanged: EventEmitter<string> = new EventEmitter();
 changeMessage() {
 this.messageChanged.emit('mono');
 }
};
```

Sử dụng decorator `@Output`, để khai báo đầu ra cho component con, với cách khai báo

**`@Output() messageChanged: EventEmitter<string> = new EventEmitter();`**

thì ở component cha, khi gọi component con, sẽ theo cấu trúc:

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

@Component({
 selector: 'parent-component',
 template: `
 <h2>Parent component</h2>
 {{message}}
 <child-component hello="truongluu"
 (messageChanged)="changeMessage($event)"></child-component>
 `
})
export class ParentComponent {
```

```

message: string = "";
changeMessage($event) {
 this.message = $event;
}
}

```

Với (messageChanged): là tên @Output đã khai báo trong component con, và trong component cha sẽ khai báo 1 hàm để nhận dữ liệu từ con phát ra

(this.messageChanged.emit('mono'), cụ thể là phát ra chuỗi "mono" khi click và button ở con. Dữ liệu được phát ra từ con cho cha là dữ liệu dạng chuỗi

(EventEmitter<string>). Trường hợp, muốn phát ra dữ liệu gì thì bạn định nghĩa trong EventEmitter<data-type>.

Cũng giống như sai @Input, thì @Output cũng cho thiết lập alias tham chiếu vào biến,

Ví dụ: ở component con ta khai báo lại

```

export class ChildComponent {
 @Input() hello: string;
 @Output('changed') messageChanged: EventEmitter<string> = new
 EventEmitter();
 changeMessage() {
 this.messageChanged.emit('mono');
 }
};

```

Lúc này, ở component cha ta sử dụng alias (changed) là output để phát ra dữ liệu cho cha

```

<child-component hello="truongluu"
(changed)="changeMessage($event)"></child-component>

```



```

<child-component hello="truongluu" (changed)="changeMessage($event)">
</child-component>

```

#### 4.7 Làm việc với @ViewChild

##### a) Khi nào chúng ta cần đến @ViewChild decorator?



Bạn muốn truy cập vào các child component, directive hay DOM element từ parent component. Việc này thật dễ dàng khi đã có ViewChild decorator.

ViewChild trả về phần tử đầu tiên mà chúng ta muốn truy vấn.

Trong trường hợp chúng ta muốn truy vấn tới nhiều phần tử con, chúng ta có thể dùng ViewChildren thay thế.

### **b) Mối quan hệ với AfterViewInit Lifecycle Hook**

Nếu chúng ta muốn component con mà chúng ta truy vấn tới thực sự được khởi tạo thì chúng ta nên thực thi trong AfterViewInit lifecycle hook

#### ***Có thể sử dụng ngOnInit() thay thế ngAfterViewInit() hay không?***

Tuỳ từng tình huống mà ta ngOnInit có thể thay thế được ngAfterViewInit khi dùng ViewChild được hay không. Nếu template được truy vấn thực sự đã sinh ra trong ngOnInit, thì chúng ta có thể dùng ngOnInit nhưng để chắc chắn và không quan tâm thời điểm template được sinh ra thì chúng ta không nên dùng ngOnInit.

### **c) Sử dụng @ViewChild để truy vấn tới DOM element**

Chúng ta có thể truy cập tới native DOM element thông qua biến tham chiếu template Chúng ta biến tham chiếu template demoInput với tình huống sau:

```
<input #demoInput placeholder="hôm nay là thu may">
```

Chúng ta có thể sử dụng ViewChild để truy cập tới input trên như sau:

```
import { Component,
 ViewChild,
 AfterViewInit,
 ElementRef } from '@angular/core';

@Component({
 selector: 'app-root',
 templateUrl: './app.component.html',
 styleUrls: ['./app.component.css']
})
export class AppComponent implements AfterViewInit {
 @ViewChild('demoInput', { static: true }) demoInput: ElementRef;

 ngAfterViewInit() {
 this.demoInput.nativeElement.value = "Sunday!";
 }
}
```

---

 }

Giá trị của Input sẽ được set thành Sunday! sau khi `ngAfterViewInit` thực thi.

#### d) Sử dụng `@ViewChild` để truy vấn tới DOM element của component

Không khó để chúng ta có thể truy cập tới component con và gọi method hoặc truy cập vào các biến có sẵn trong component con.

Chúng ta có hàm `whoAmI` như sau viết trong component con:

```
whoAmI() {
 return 'I am a grown-up sunner!';
}
```

Chúng ta có thể gọi đến method đó từ component cha khi sử dụng `ViewChild` như sau:

```
import { Component,
 ViewChild,
 AfterViewInit } from '@angular/core';
import { ChildComponent } from './child.component';
@Component({
 selector: 'app-root',
 templateUrl: './app.component.html',
 styleUrls: ['./app.component.css']
})
export class AppComponent implements AfterViewInit {
 @ViewChild(ChildComponent, { static: false }) child: ChildComponent;
 ngAfterViewInit() {
 console.log(this.child.whoAmI()); // I am a grown-up sunner!
 }
}
```

**Chú ý:** Nếu giá trị `static` là `true`, angular sẽ cố tìm kiếm phần tử từ thời điểm khởi tạo component, ví dụ `ngOnInit`. Nó có thể hoàn thành khi phần tử không được sử dụng bất kỳ structure directive nào. Khi `static value` là `false`, angular sẽ tìm phần tử sau khi khởi tạo view.

## Bài 5: Template và Data Binding(tiếp)

### 5.1 Directive là gì?

Directives có thể hiểu như là các đoạn mã typescript (hoặc javascript) kèm theo cả html và khi gọi thì gọi như là html luôn, ví dụ:

```
<div *ngIf="title"> <!-- Chỗ này là gọi directive ngIf để kiểm tra điều kiện if ngay ở html -->
 Time: {{title}}
</div>
```

Từ Angular 2 trở đi, directives được chia làm các loại sau đây:

- Components: Không có nghi ngờ gì khi gọi component là directive cũng được, vì rõ ràng là component cho phép định nghĩa selector và gọi ra như một thẻ html tag (<component-name></component-name>)
- Structural directives: Là directive cấu trúc, dùng để vẽ html, hiển thị data lên giao diện html. Ví dụ ngFor, ngIf
- Attribute directives: Thêm các thuộc tính động cho element html, ví dụ ngStyle

### 5.2 Structure Directive: ngIf, ngFor, ngSwitch, Ng-Template, Ng-Container

#### a) ngIf

Chỉ thị ngIf cho phép chúng ta thêm hoặc loại bỏ một element ra khỏi trang, chúng ta gán giá trị cho chỉ thị này là một biểu thức nào đó có trả về giá trị true hoặc false, nếu biểu thức trả về true thì element sẽ hiện ra, ngược lại thì không.

```
<customer *ngIf="isActive"></customer>
```

Trong đoạn code trên, element <customer> sẽ được hiển thị nếu isActive trả về true, isActive có thể là một thuộc tính/biến nào đó hoặc một phương thức...v.v

**Lưu ý:** Luôn phải có dấu sao \* trước ngIf.

Một điều khác là ở đây ngIf thêm hoặc bỏ element trong trang web, chứ không phải là ẩn hay hiện element đó, tức là khác với thuộc tính hidden của các element trong HTML.

Một ví dụ khác:

```
<div *ngIf="title; else noTitle">
 Time: {{title}}
</div>

<ng-template #noTitle> Click on the button to see time. </ng-template>
```

Code ở trên, khi biến title có giá trị, thì chuỗi Time: [value] được show ra. Và cục #noTitle template bị ẩn đi, ngược lại thì điều kiện else được chạy và #noTitle được hiện ra.

Như ta thấy dùng cái directive ngIf else này rất tiện lợi khi có thể ẩn hiện html dễ dàng.

## b) ngFor

Đây là chỉ thị lặp, có tác dụng lặp qua một danh sách các phần tử, khi chúng ta có một danh sách các phần tử, muốn hiển thị chúng lên trang web thì chúng ta lặp qua danh sách đó và hiển thị các phần tử theo một khuôn mẫu giống nhau. Ví dụ:

```
<div *ngFor="let cus of customers">
 {{cus.name}}
</div>
```

Giá trị của ngFor là một câu lệnh có cú pháp như sau:

*let <biến lặp> of <danh sách>*

Biến lặp là do chúng ta tự đặt, bạn muốn đặt là gì cũng được, ngFor sẽ lặp qua danh sách và mỗi lần lặp thì chúng ta dùng biến lặp để lấy dữ liệu của phần tử hiện tại trong danh sách.

Lưu ý: Luôn phải có dấu sao \* trước ngFor.

Trong ngFor có một thuộc tính tên là index, thuộc tính này lưu trữ số thứ tự của phần tử đang được lặp, chúng ta có thể lấy số thứ tự giá trị index này như sau:

```
<div *ngFor="let cus of customers; let i=index">{{i + 1}} : {{cus.name}}</div>
```

## c) ngSwitch

Chúng ta hoàn toàn có thể sử dụng câu lệnh điều kiện switch case trong Angular y như switch case trong Javascript vậy.

Cách viết như sau:

```
<div [ngSwitch]="isMetric">
 <div *ngSwitchCase="true">Degree Celsius</div>
 <div *ngSwitchCase="false">Fahrenheit</div>
</div>
```

Trong trường hợp muốn dùng switch case default (nếu toàn bộ case k thỏa mãn thì vào default) thì chúng ta viết như sau:

```
<div [ngSwitch]="isMetric">
 <div *ngSwitchCase="true">Degree Celsius</div>
 <div *ngSwitchDefault>Fahrenheit</div>
</div>
```

#### d) Ng-Template

Đây cũng là một Structural directives. Nó giúp gom cục html cần ẩn hiện.

```
<div *ngIf="isTrue; then tmplWhenTrue else tmplWhenFalse"></div>
<ng-template #tmplWhenTrue >I show-up when isTrue is true. </ng-template>
<ng-template #tmplWhenFalse > I show-up when isTrue is false </ng-template>
```

Chú ý là đoạn code vừa rồi dùng ngIf..then..else.

#### e) Ng-Container

Tương tự như Ng-Template dùng để gom html. Nhưng điểm mạnh của Ng-Container là thẻ directive này không render ra tag <ng-container> html như là <ng-template> mà tag sẽ được ẩn đi, giúp cho layout css không bị vỡ nếu bạn gom html (Không sợ bị nhảy từ div cha sang div con, cấu trúc html k hề thay đổi khi gom vào tag <ng-container></ng-container>)

Hãy xem ví dụ sau đây:

```
Welcome <div *ngIf="title">to <i>the</i> {{title}} world.</div>
```

Sẽ được render ra như sau:

Welcome  
to the Angular world.

Khi soi html chúng ta sẽ thấy:

```
"
Welcome "
<!--bindings={
 "ng-reflect-ng-if": "Angular"
}-->
<div _ngcontent-c0>
 "to "
 <i _ngcontent-c0>the</i>
 " Angular world."
</div>
<hr _ngcontent-c0>
```

Tự dung dòng div có ngIf nó lại chèn một cái thuộc tính \_ngcontent-c0, dẫn đến dòng đó bị xuống dòng, làm sai layout design.

Bây giờ hãy viết lại như sau:

Welcome <ng-container \*ngIf="title">to <i>the</i> {{title}} world.</ng-container>

Kết quả sẽ như sau:

Welcome to *the* Angular world.

Đó là vì html đã được dọn gọn gàng:

```

Welcome "
<!--bindings={
 "ng-reflect-ng-if": "Angular"
}-->
<!-->
"to "
<i _ngcontent-c0>the</i>
" Angular world."

```

### 5.3 Attribute Directive: ngStyle, ngClass, ngContainer

#### a) ngStyle

Thuộc tính ngStyle được sử dụng để thay đổi thuộc tính của một thẻ HTML. Bạn có thể thay đổi giá trị, màu, và kích thước, vv của các yếu tố.

Sau đây là một ví dụ thể hiện điều kiện ngStyle

Ví dụ 1:

```
<div [ngStyle]="{'background-color':'green'}"></div>
```

Ví dụ 2:

```
<div [ngStyle]="{'background-color':person.country === 'UK' ? 'green' : 'red'}"></div>
```

Ví dụ 3:

```

@Component({
 selector: 'ngstyle-example',
 template: `<h4>NgStyle</h4>
<ul *ngFor="let person of people">
 <li [ngStyle]="{'color':getColor(person.country)}"> {{ person.name }} ({{
 person.country }}) (1)

`
})
class NgStyleExampleComponent {

```

```
getColor(country) { (2)
 switch (country) {
 case 'UK':
 return 'green';
 case 'USA':
 return 'blue';
 case 'HK':
 return 'red';
 }
}
people: any[] = [
 {
 "name": "Douglas Pace",
 "country": 'UK'
 },
 {
 "name": "Mcleod Mueller",
 "country": 'USA'
 },
 {
 "name": "Day Meyers",
 "country": 'HK'
 },
 {
 "name": "Aguirre Ellis",
 "country": 'UK'
 },
 {
 "name": "Cook Tyson",
 "country": 'USA'
 }
]
```

```
];
```

```
}
```

Ví dụ 4:

```
<ul *ngFor="let person of people">
 <li [style.color]="getColor(person.country)">{{ person.name }} ({{
 person.country }})


```

#### a) ngClass

Trong bài phần trước, chúng ta đã thấy rằng làm thế nào để sử dụng `ngStyle` để thực hiện thay đổi trong một yếu tố động. Ở đây, chúng ta sẽ sử dụng chỉ thị `ngClass` để áp dụng một lớp CSS cho phần tử. Nó tạo điều kiện cho bạn để thêm hoặc loại bỏ một CSS động.

Ví dụ 1:

```
[ngClass]="{'text-success':true}"
```

Ví dụ 2:

```
[ngClass]="{'text-success':person.country === 'UK'}"
```

Ví dụ 3:

```
<h4>NgClass</h4>
<ul *ngFor="let person of people">
 <li [ngClass]="{
 'text-success':person.country === 'UK',
 'text-primary':person.country === 'USA',
 'text-danger':person.country === 'HK'
 }">{{ person.name }} ({{ person.country }})


```

Ví dụ 4:

```
[class.text-success]="true"
```

Ví dụ 5:

```
<ul *ngFor="let person of people">
 <li [class.text-success]="person.country === 'UK'"
```



```
[class.text-primary]="person.country === 'USA'"
[class.text-danger]="person.country === 'HK'">{{ person.name }} ({{
person.country }})


```

## 5.4 Pipes

### a) Giới thiệu pipe

Pipe là một tính năng được xây dựng sẵn từ Angular 2 với mục tiêu nhằm biến đổi dữ liệu đầu ra, hiển thị lên trên template đúng với ý tưởng thiết kế lập trình, thân thiện với người sử dụng .

Ví dụ là định dạng kiểu hiển thị datetime, viết hoa chữ cái, hiển thị tên thành phố, định dạng lại số hay đơn vị tiền, ...

### b) Sử dụng pipe

Ví dụ dưới đây sẽ sử dụng pipe để biến đổi ngày sinh nhật từ dữ liệu thô sang định dạng dễ đọc hơn

```
import { Component } from '@angular/core';
@Component({
 selector: 'app-hero-birthday',
 template: `<p>The hero's birthday is {{ birthday | date }}</p>`
})
export class HeroBirthdayComponent {
 birthday = new Date(1988, 3, 15);
}
```

Kết quả: Fri Apr 15 1988 00:00:00 GMT+0700 (Indochina Time) -> April 15, 1988

Bên trong biểu thức nội suy `{{ }}`, bạn đã biến đổi ngày sinh nhật bằng biểu thức pipe | từ định dạng của js Fri Apr 15 1988 00:00:00 GMT+0700 (Indochina Time) sang April 15, 1988, dễ đọc cho người dùng

### c) Tham số trong pipe

Để có thể bổ sung tham số trong pipe, chúng ta sử dụng dấu hai chấm : sau tên pipe đó (ví dụ `currency:'EUR'`)

Nếu pipe đó chấp nhận nhiều tham số đúng với cú pháp, hay viết : nằm giữa hai tham số (ví dụ slice:1:5)

```
<p>The hero's birthday is {{ birthday | date:"MM/dd/yy" }} </p>
```

Tham số có thể được custom lại thông qua các thao tác thay đổi sự kiện (nhưng vẫn phải đúng với cú pháp của pipe đó)

Sử dụng tiếp với ví dụ trên, chúng ta sẽ không truyền tham số định dạng trực tiếp đằng sau datepipe mà sẽ ràng buộc thông qua một biến là format và chúng ta sẽ thay đổi format đó thông qua một button bên dưới

```
<p>The hero's birthday is {{ birthday | date:format }}</p>
```

```
<button (click)="toggleFormat()">Toggle Format</button>
```

Sự kiện click sẽ thay đổi format của datepipe giữa kiểu shortDate và fullDate

```
export class HeroBirthday2Component {
 birthday = new Date(1988, 3, 15); // April 15, 1988
 toggle = true; // start with true == shortDate
 get format() { return this.toggle ? 'shortDate' : 'fullDate'; }
 toggleFormat() { this.toggle = !this.toggle; }
}
```

#### d) Chuỗi các pipe

Bạn có thể kết hợp sử dụng các pipe liên tiếp với nhau, pipe trước sẽ là dữ liệu đầu vào của pipe sau, đúng như nghĩa đen của nó là ống nước

Ví dụ, sử dụng kết hợp cả DatePipe và UpperCasePipe

```
{{ birthday | date | uppercase }}
// APR 15, 1988
```

Sử dụng tham số cùng với chuỗi pipe

```
{{ birthday | date:'fullDate' | uppercase }}
// FRIDAY, APRIL 15, 1988
```

#### e) Custom pipes

Ngoài những pipe mà angular tích hợp sẵn trong code, nó còn cung cấp cho chúng ta công cụ để có thể tự sáng tạo ra những pipe của riêng mình

#### exponential-strength.pipe.ts

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

/*
```

```

* Raise the value exponentially
* Takes an exponent argument that defaults to 1.
* Usage:
* value / exponentialStrength:exponent
* Example:
* {{ 2 / exponentialStrength:10 }}
* formats to: 1024
*/

```

```

@Pipe({name: 'exponentialStrength'})
export class ExponentialStrengthPipe implements PipeTransform {
 transform(value: number, exponent?: number): number {
 return Math.pow(value, isNaN(exponent) ? 1 : exponent);
 }
}

```

Pipe trên là một custom pipe

Có thể hiểu đầu vào là 2 tham số, tham số thứ nhất là số ngẫu nhiên, tham số thứ 2 là số mũ. Đầu ra sẽ là phép tính lũy thừa từ hai số

```

import { Component } from '@angular/core';
@Component({
 selector: 'app-power-booster',
 template: `
 <h2>Power Booster</h2>
 <p>Super power boost: {{2 / exponentialStrength: 10}}</p>
 `,
})
export class PowerBoosterComponent { }

```

Kết quả:

## Power Booster

Super power boost: 1024

Những nội dung cần làm để tạo một custom pipe

- Viết một class và decorate nó với decorator `@Pipe` ( báo cho angular biết đây là pipe) và decorator này cần tối thiểu một object có property name, chính là tên của pipe để có thể gọi ra trong template.
- Tiếp theo, chúng ta cần implement một interface là `PipeTransform`, và implement hàm transform của interface đó. Hàm transform sẽ làm biến đổi dữ liệu
- Sau khi hoàn thiện pipe trên, ta cần import custom pipe trên vào trong `NgModule` mà template cần sử dụng custom pipe thuộc về ( ví dụ trên là viết cùng file )

Ví dụ trên là dữ liệu tĩnh, bây giờ chúng ta sẽ làm cho nó động hơn với `ngModel`

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

@Component({
 selector: 'app-power-boost-calculator',
 template: `
 <h2>Power Boost Calculator</h2>
 <div>Normal power: <input [(ngModel)]="power"></div>
 <div>Boost factor: <input [(ngModel)]="factor"></div>
 <p>
 Super Hero Power: {{power | exponentialStrength: factor}}
 </p>
 `,
})
export class PowerBoostCalculatorComponent {
 power = 5;
 factor = 1;
}
```

Kết quả:

## Power Boost Calculator

Normal power:   
 Boost factor:   
 Super Hero Power: 5

## Power Boost Calculator

Normal power:   
 Boost factor:   
 Super Hero Power: 32

### 5.5 Animation

Animation thêm rất nhiều tương tác giữa các thẻ html. Animation thì đã có từ Angular 2. Điểm khác biệt ở Angular 4 là animation được tách thành một package riêng biệt chứ không còn nằm trong package `@angular/core` nữa.

Để bắt đầu sử dụng animation, chúng ta cần import thư viện từ package

`@angular/platform-browser/animations`

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

Cũng như các module khác, chúng ta cần khai báo bên trong mảng import

**app.module.ts**

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-
browser/animations';
import { AppComponent } from './app.component';
@NgModule({
 declarations: [
 AppComponent
],
 imports: [
 BrowserModule,
 BrowserModule
],
 providers: [],
 bootstrap: [AppComponent]
})
export class AppModule { }
```

Tiếp đó, trong file `app.component.html`, chúng ta thêm các phần tử html để áp dụng animation lên

#### **app.component.html**

```
<div>
 <button (click)="animate()">Click Me</button>
 <div [@myanimation] = "state" class="rotate">

 </div>
</div>
```

Trong thẻ `div` chính, chúng ta có 1 button và `div` con kèm 1 ảnh. Button được gán sự kiện `click`; `div` con được gán sử dụng `@myanimation` directive với giá trị là `state`.

Trong file `app.component.ts`, chúng ta định nghĩa animation

#### **app.component.ts**

```
import { Component } from '@angular/core';
import { trigger, state, style, transition, animate } from '@angular/animations';
@Component({
 selector: 'app-root',
 templateUrl: './app.component.html',
 styleUrls: ['./app.component.css'],
 styles: [`
 div{
 margin: 0 auto;
 text-align: center;
 }
 .rotate{
 width: 340px;
 height: 82px;
 border:solid 1px red;
 }
 `],
})
```

```

 animations: [
 trigger('myanimation', [
 state('smaller', style({
 transform: 'translateY(100px)'
 })),
 state('larger', style({
 transform: 'translateY(0px)'
 })),
 transition('smaller <=> larger', animate('300ms ease-in'))
])
]
 })

export class AppComponent {
 state: string = "smaller";
 animate() {
 this.state = this.state == 'larger' ? 'smaller' : 'larger';
 }
}

```

Ở dòng code thứ 2, chúng ta import các class cần thiết cho animation: trigger, state, style, transition, animate:

import { trigger, state, style, transition, animate } from '@angular/animations';

Bên trong decorator @Component, ta định nghĩa thêm thuộc tính animations:

```

 animations: [
 trigger('myanimation', [
 state('smaller', style({
 transform: 'translateY(100px)'
 })),
 state('larger', style({
 transform: 'translateY(0px)'
 })),
 transition('smaller <=> larger', animate('300ms ease-in'))
])
]
 })

```

]

Trigger định nghĩa rằng animation được sử dụng như thế nào. Tham số đầu tiên chính là tên của animation mà ta sẽ sử dụng bên trong thẻ html. Trong ví dụ trên thì đó là 'myanimation'. Tham số thứ 2 chứa các hàm state và transition.

Hàm state định nghĩa các trạng thái của animation: với trạng thái nào thì sẽ có style như thế nào. Trong ví dụ trên thì ta định nghĩa 2 state là: smaller và larger. State smaller có style là transform: translateY(100px) - tức dịch chuyển 100px theo trục y trong toạ độ đề-các và state larger có style là transform: translateY(0px) - tức dịch chuyển lại vị trí ban đầu.

Hàm transition định nghĩa các thông số hiển thị trên màn hình như độ dài, độ trễ khi chuyển từ trạng thái này sang trạng thái khác.

Cùng nhìn vào file html để xem transition hoạt động như thế nào nhé

```
<div>
 <button (click)="animate()">Click Me</button>
 <div [@myanimation] = "state" class="rotate">

 </div>
</div>
```

Thuộc tính style được thêm vào @component directive để căn giữa cho các thẻ div.

```
styles: [`
 div{
 margin: 0 auto;
 text-align: center;
```



```

 }
 .rotate{
 width: 340px;
 heigh: 82px;
 border:solid 1px red;
 }
 },

```

Khi click vào button, hàm animate được khai báo trong file app.component.ts được gọi tới

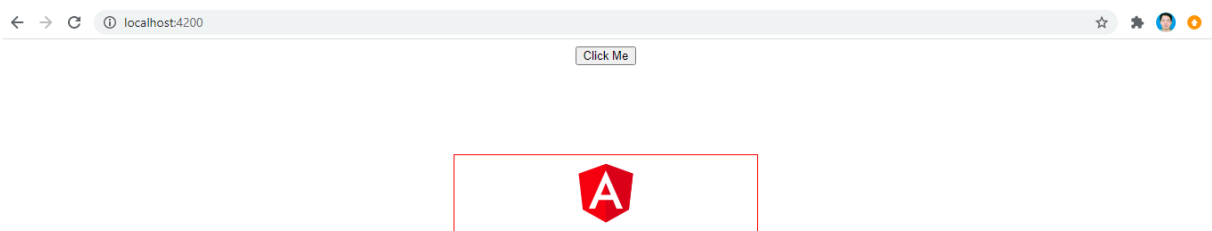
```

state: string = "smaller";
animate() {
 this.state= this.state == 'larger' ? 'smaller' : 'larger';
}

```

Biến state được định nghĩa giá trị mặc định là smaller. Nếu state đang là larger thì chuyển về smaller và ngược lại.

Khi mở trình duyệt lên, ta nhận được kết quả



Khi click vào button Click Me, ta nhận được kết quả như hình



## Bài 6: Routing & Navigation

### 6.1 Cơ bản về routing

Angular cung cấp cho bạn khả năng chia ứng dụng thành các view riêng biệt và bạn có thể điều hướng giữa các view thông qua routing. Nó cho phép bạn điều hướng người dùng tới các component khác nhau dựa trên url.

Nếu bạn mở file *systemjs.config.js* bạn sẽ thấy router trong số các packages của Angular:

```
var ngPackageNames = [
 'common',
 'compiler',
 'core',
 'forms',
 'http',
 'platform-browser',
 'platform-browser-dynamic',
 'router', // <===== here!
 'router-deprecated',
 'upgrade',
];
```

Nếu bạn đã làm việc với Angular 1 và *ngRoute* bạn sẽ thấy phần còn lại của bài viết này khá quen thuộc, điểm khác biệt lớn nhất là thay vì mapping các *route* với các *controller*, Angular map các *route* tới các *component*, và Angular thiên về khai báo nhiều hơn.

#### Thiết lập route people list

Angular cấu hình các route thông qua một file cấu hình gồm:

- Một mảng Routes sẽ chứa tập hợp các route.
- Một export cung cấp router tới phần còn lại của ứng dụng.

Chúng ta sẽ bắt đầu với việc tạo file *app.routes.ts* trong thư mục app và *import* interface Routes từ module *@angular/router*:

```
import { Routes } from '@angular/router';
```

Bây giờ, chúng ta sẽ định nghĩa route mặc định của ứng dụng là danh sách các nhân vật Star Wars. Trước khi làm việc đó chúng ta cần *import* *PeopleListComponent*:

```
import { PeopleListComponent } from './people-list.component';
```

Và tạo một mảng Routes:

```
// Route config let's you map routes to components
const routes: Routes = [
 // map '/persons' to the people list component
 {
 path: 'persons',
 component: PeopleListComponent,
 },
 // map '/' to '/persons' as our default route
 {
 path: '',
 redirectTo: '/persons',
 pathMatch: 'full'
 },
];
```

Như bạn thấy, route đầu tiên map path *persons* tới component *PeopleListComponent*. Chúng ta nói với Angular rằng, đây là route mặc định bằng cách thêm một cấu hình, map với một path rỗng và redirect tới path *persons*.

Bước tiếp theo là làm cho routes của chúng ta có thể sử dụng trong phần còn lại của ứng dụng. Trước hết, chúng ta cần import *RouterModule* từ module *@angular/router*:

```
import { Routes, RouterModule } from '@angular/router';
```

Và export các route đã định nghĩa như sau:

```
export const routing = RouterModule.forRoot(routes);
```

Toàn bộ file cấu hình route *app.routes.ts* sẽ như thế này:

```
import { Routes, RouterModule } from '@angular/router';
import { PeopleListComponent } from './people-list.component';
// Route config let's you map routes to components
const routes: Routes = [
 // map '/persons' to the people list component
 {
 path: 'persons',
```

```
 component: PeopleListComponent,
 },
 // map '/' to '/persons' as our default route
 {
 path: "",
 redirectTo: '/persons',
 pathMatch: 'full'
 },
];

export const routing = RouterModule.forRoot(routes);
```

Bây giờ, chúng ta có thể cập nhật ứng dụng để sử dụng các route đã định nghĩa trong file cấu hình. Hãy import routing trong file *app.module.ts*:

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { routing } from './app.routes';
import { AppComponent } from './app.component';
import { PeopleListComponent } from './people-list.component';
import { PersonDetailsComponent } from './person-details.component';
@NgModule({
 imports: [BrowserModule, routing],
 declarations: [AppComponent, PeopleListComponent, PersonDetailsComponent],
 bootstrap: [AppComponent]
})
export class AppModule { }
```

Cần lưu ý, mặc dù bạn thường định nghĩa routing ở mức ứng dụng trong file cấu hình, nhưng bạn cũng có thể làm điều đó ở mức component để tối đa hóa khả năng tái sử dụng và độc lập.

Mặc dù, đã cung cấp các thông tin về route cho ứng dụng, chúng ta vẫn sử dụng component `<people-list>` trực tiếp trong template của AppComponent. Cái chúng ta thực sự muốn là lựa chọn component được hiển thị dựa trên đường dẫn trong trình duyệt. Để làm điều đó, chúng ta sử dụng *router-outlet* directive, một Angular Routing directive để hiển thị route active (giống ng-view):

```
<h1>{{title}}</h1>
```

```
<router-outlet></router-outlet>
```

Toàn bộ AppComponent sẽ như thế này:

```
import { Component } from '@angular/core';
import { PeopleService } from './people.service';
@Component({
 selector: 'my-app',
 template: `
 <h1> {{title}} </h1>
 <router-outlet></router-outlet>
 `,
 providers: [PeopleService]
})
export class AppComponent {
 title:string = 'Star Wars Peoplez!';
}
```

Nếu bạn gõ lệnh `npm start`, bạn sẽ thấy app bị lỗi. Sử dụng dev tools của trình duyệt bạn sẽ thấy thông báo lỗi trong console như sau:

```
Subscriber.ts:243 Uncaught EXCEPTION: Error during instantiation of
LocationStrategy! (Router -> Location -> LocationStrategy).
ORIGINAL EXCEPTION: No base href set. Please provide a value for the
APP_BASE_HREF token or add a base element to the document.
```

Bạn chỉ cần bổ sung thêm thẻ base trong phần head của file `index.html` như dưới đây:

```
<html>
 <head>
 <title>Angular QuickStart</title>
 <base href="/">
 <!-- etc... -->
 </head>
 <!-- etc... -->
</html>
```

Thẻ này cho phép api *history.pushState* của HTML5, giúp Angular cung cấp các URL không có tiền tố #.

Bây giờ nếu gõ lệnh *npm start* một lần nữa bạn sẽ thấy ứng dụng làm việc. Và url trên trình duyệt sẽ như thế này:

```
http://localhost:3000/persons
```

Điều này có nghĩa là routing đã làm việc như mong đợi.

### Thiết lập person details route

Bây giờ, chúng ta sẽ thiết lập person details route và thay đổi workflow của ứng dụng, chúng ta sẽ hiển thị thông tin chi tiết trong một view hoàn toàn khác.

Chúng ta bắt đầu bằng việc import *PersonDetailsComponent* và định nghĩa một route mới trong *app.routes.ts*:

```
import { Routes, RouterModule } from '@angular/router';
import { PeopleListComponent } from './people-list.component';
// import PersonDetailsComponent
import { PersonDetailsComponent } from './person-details.component';
// Route config let's you map routes to components
const routes: Routes = [
 // map '/persons' to the people list component
 {
 path: 'persons',
 component: PeopleListComponent,
 },
 // map '/persons/:id' to person details component
 {
 path: 'persons/:id',
 component: PersonDetailsComponent
 },
 // map '/' to '/persons' as our default route
 {
 path: '',
 redirectTo: '/persons',
 pathMatch: 'full'
 }
];
```

```
},
];

export const routing = RouterModule.forRoot(routes);
```

Chúng ta đã định nghĩa route map `/persons/:id` tới `PersonDetailsComponent`. `:id` là tham số, được sử dụng để xác định nhân vật sẽ hiển thị thông tin chi tiết.

Điều đó có nghĩa là, chúng ta cần cập nhật thuộc tính `id` trong interface `Person` như sau:

```
export interface Person {
 id: number;

 name: string;

 height: number;

 weight: number;
}
```

Và service `PeopleService` cũng cần cập nhật thêm thuộc tính `id` trong mảng `PEOPLE`:

```
import { Injectable } from '@angular/core';
import { Person } from './person';

const PEOPLE : Person[] = [
 {id: 1, name: 'Luke Skywalker', height: 177, weight: 70},
 {id: 2, name: 'Darth Vader', height: 200, weight: 100},
 {id: 3, name: 'Han Solo', height: 185, weight: 85},
];

@Injectable()
export class PeopleService{
 getAll() : Person[] {
 return PEOPLE;
 }
}
```

### Tạo các liên kết route

Bây giờ, chúng ta đã định nghĩa route nhưng chúng ta muốn người dùng có thể truy cập tới trang hiển thị thông tin chi tiết khi họ click vào một nhân vật trong view hiển thị danh sách các nhân vật. Làm thế nào chúng ta có thể làm điều đó?

Angular cung cấp directive `[routerLink]` giúp bạn tạo ra các liên kết cực kỳ đơn giản.

Chúng ta sẽ cập nhật template của *PeopleListComponent* để sử dụng *[routerLink]* thay vì event (click). Chúng ta cũng cần xóa phần tử *<person-details>* trong template.

```

 <li *ngFor="let person of people">

 {{person.name}}


```

Trong phần source code phía trên bạn có thể thấy cách chúng ta liên kết một mảng các tham số route với directive *routerLink*, một tham số là đường dẫn của route */persons* và tham số còn lại là id thực sự. Với 2 tham số này Angular có thể tạo ra các liên kết phù hợp (VD: */person/2*).

Chúng ta cũng cần xóa bỏ một vài đoạn code không cần thiết như các hàm xử lý sự kiện (click) và thuộc tính *selectedPerson*. *PeopleListComponent* sau khi cập nhật sẽ như thế này:

```
import { Component, OnInit } from '@angular/core';
import { Person } from './person';
import { PeopleService } from './people.service';

@Component({
 selector: 'people-list',
 template: `
 <!-- this is the new syntax for ng-repeat -->

 <li *ngFor="let person of people">

 {{person.name}}

 `,
})
```



```

export class PeopleListComponent implements OnInit{
 people: Person[] = [];
 constructor(private peopleService : PeopleService){ }
 ngOnInit(){
 this.people = this.peopleService.getAll();
 }
}

```

Ok, bây giờ hãy quay trở lại trình duyệt. Nếu bạn di chuột lên tên của mỗi nhân vật bạn sẽ nhìn thấy nó trở tới liên kết chính xác.

Nếu bạn click vào tên nhân vật nó sẽ không làm việc. Bởi vì *PersonDetailsComponent* không biết cách nhận id từ route, nhận một nhân vật với id và hiển thị nó trong view.

Tiếp theo, hãy làm điều đó!

### Tách các tham số từ routes

Trong phần trước *PersonDetailsComponent* sử dụng thuộc tính *person* nhận dữ liệu về nhân vật được chọn. Khi sử dụng route chúng ta sẽ không làm như vậy.

Chúng ta sẽ cập nhật *PersonDetailsComponent* để tách thông tin từ route, nhận dữ liệu phù hợp sử dụng *PeopleService* và hiển thị nó.

Angular routing cung cấp service *ActivateRoute* cho mục đích này, lấy các tham số route. Chúng ta có thể import nó từ module *@angular/router* và thêm nó

vào *PersonDetailsComponent* thông qua hàm khởi tạo:

```

import { ActivatedRoute } from '@angular/router';
export class PersonDetailsComponent{
 constructor(private peopleService: PeopleService,
 private route: ActivatedRoute){
 }
 // more codes...
}

```

Bây giờ chúng ta có thể sử dụng nó để nhận tham số id từ url và lấy thông tin về nhân vật từ *PeopleService*. Chúng ta sẽ làm điều đó trong *ngOnInit*:

```

export class PersonDetailsComponent implements OnInit {
 person: Person;

```

```
// more codes...
ngOnInit(){
 this.route.params.subscribe(params => {
 let id = Number.parseInt(params['id']);
 this.person = this.peopleService.get(id);
 });
}
```

Chú ý, *route.params* trả lại một *observable*, một pattern để xử lý các thao tác bất động bộ. Phương thức *subscribe* giúp chúng ta nạp một route và nhận các tham số.

Để tránh memory leaks chúng ta có thể sử dụng *unsubscribe* cho observable *route.params* khi Angular destroy *PersonDetailsComponent*. Chúng ta có thể tận dụng lợi thế của *onDestroy*:

```
import { Component, OnInit, OnDestroy } from '@angular/core';
export class PersonDetailsComponent implements OnInit {
 // more codes...
 sub: any;
 ngOnInit(){
 this.sub = this.route.params.subscribe(params => {
 let id = Number.parseInt(params['id']);
 this.person = this.peopleService.get(id);
 });
 }
 ngOnDestroy(){
 this.sub.unsubscribe();
 }
}
```

*PersonDetailsComponent* khi hoàn thành sẽ như thế này:

```
import { Component, OnInit, OnDestroy } from '@angular/core';
import { ActivatedRoute } from '@angular/router';
import { Person } from './person';
import { PeopleService } from './people.service';
```

```
@Component({
 selector: 'person-details',
 template: `
 <!-- new syntax for ng-if -->
 <section *ngIf="person">
 <h2>You selected: {{person.name}} </h2>
 <h3>Description</h3>
 <p>
 {{person.name}} weighs {{person.weight}} and is {{person.height}} tall.
 </p>
 </section>
 `,
})

export class PersonDetailsComponent implements OnInit, OnDestroy {
 person: Person;
 sub: any;
 constructor(private peopleService: PeopleService,
 private route: ActivatedRoute){
 }
 ngOnInit(){
 this.sub = this.route.params.subscribe(params => {
 let id = Number.parseInt(params['id']);
 this.person = this.peopleService.get(id);
 });
 }
 ngOnDestroy(){
 this.sub.unsubscribe();
 }
}
```

Chúng ta cũng cần cập nhật *PeopleService* để cung cấp một phương thức mới để lấy dữ liệu về một nhân vật bằng id, phương thức `get(id: number)` dưới đây sẽ làm điều đó:

```

import { Injectable } from '@angular/core';
import { Person } from './person';
const PEOPLE : Person[] = [
 {id: 1, name: 'Luke Skywalker', height: 177, weight: 70},
 {id: 2, name: 'Darth Vader', height: 200, weight: 100},
 {id: 3, name: 'Han Solo', height: 185, weight: 85},
];
@Injectable()
export class PeopleService{
 getAll() : Person[] {
 return PEOPLE;
 }
 get(id: number) : Person {
 return PEOPLE.find(p => p.id === id);
 }
}

```

Chúng ta đã hoàn thành! Bây giờ nếu click vào một nhân vật, bạn sẽ thấy thông tin chi tiết về nhân vật đó.

Nhưng làm thế nào bạn có thể trở lại view chính (view danh sách các nhân vật). Bạn có thể làm điều đó bằng cách nhấn nút back của trình duyệt. Nhưng nếu bạn muốn có một nút back ở phía dưới thông tin chi tiết thì sao?

### Trở lại danh sách các nhân vật

Bây giờ, chúng ta muốn tạo ra một nút cho phép người dùng trở lại view chính khi anh/cô ấy click vào nó. Angular Routing cung cấp service Router giúp bạn làm điều đó. Chúng ta bắt đầu bằng cách thêm nút vào template và liên kết nó với phương thức `goToPeopleList`:

```

<section *ngIf="person">
 <h2>You selected: {{person.name}}</h2>
 <h3>Description</h3>
 <p>
 {{person.name}} weights {{person.weight}} and is {{person.height}} tall.
 </p>

```

---

```
</section>
```

```
<! -- NEW BUTTON HERE! -->
```

```
<button (click)="gotoPeoplesList()">Back to peoples list</button>
```

Chúng ta có thể tiêm service này vào *PersonDetailsComponent* thông qua hàm khởi tạo. Chúng ta bắt đầu bằng cách import nó từ *@angular/router*:

```
import { ActivatedRoute, Router } from '@angular/router';
```

Và sau đó tiêm nó:

```
export class PersonDetailsComponent implements OnInit, OnDestroy {
 // other codes...
 constructor(private peopleService: PeopleService,
 private route: ActivatedRoute,
 private router: Router){
 }
}
```

Tiếp theo là phương thức *goToPeopleList* để điều hướng trở lại view chính:

```
export class PersonDetailsComponent implements OnInit, OnDestroy {
 // other codes...
 gotoPeoplesList(){
 let link = ['/persons'];
 this.router.navigate(link);
 }
}
```

Chúng ta gọi phương thức *router.navigate* và truyền cho nó các tham số cần thiết giúp Angular routing xác định nơi chúng ta muốn tới. Trong trường hợp này, route không yêu cầu bất kỳ tham số nào, chúng ta chỉ cần tạo một mảng với đường dẫn */persons*. Nếu bạn trở lại trình duyệt và kiểm tra, bạn sẽ thấy mọi thứ hoạt động đúng như mong đợi. Bây giờ, bạn đã học Angular routing.

Đây là source code đầy đủ của *PersonDetailsComponent*:

```
import { Component, OnInit, OnDestroy } from '@angular/core';
import { ActivatedRoute, Router } from '@angular/router';
import { Person } from './person';
import { PeopleService } from './people.service';
```

```
@Component({
 selector: 'person-details',
 template: `
 <!-- new syntax for ng-if -->
 <section *ngIf="person">
 <h2>You selected: {{person.name}} </h2>
 <h3>Description</h3>
 <p>
 {{person.name}} weighs {{person.weight}} and is {{person.height}} tall.
 </p>
 <!-- NEW BUTTON HERE! -->
 <button (click)="gotoPeoplesList()">Back to peoples list</button>
 </section>
 `,
})
export class PersonDetailsComponent implements OnInit, OnDestroy {
 person: Person;
 sub: any;
 constructor(private peopleService: PeopleService,
 private route: ActivatedRoute,
 private router: Router){
 }
 ngOnInit(){
 this.sub = this.route.params.subscribe(params => {
 let id = Number.parseInt(params['id']);
 this.person = this.peopleService.get(id);
 });
 }
 ngOnDestroy(){
 this.sub.unsubscribe();
 }
}
```

```
gotoPeoplesList(){
 let link = ['/persons'];
 this.router.navigate(link);
}
}
```

Một tùy chọn khác là bạn có thể sử dụng api `window.history` trong phương thức `goToPeopleList` như bên dưới:

```
gotoPeoplesList(){
 window.history.back();
}
```

## 6.2 Lazy Loading

### Sơ qua về Lazy Loading

**Lazy Loading** là một design pattern thường được sử dụng trong lập trình máy tính để trì hoãn lại việc khởi tạo một đối tượng cho đến khi nào nó thực sự cần đến. Nó góp phần giúp cho hoạt động của chương trình được hiệu quả hơn nếu như được sử dụng một cách hợp lý. Nói đơn giản là: *"Không load bất kỳ thứ gì nếu như bạn không cần đến"*

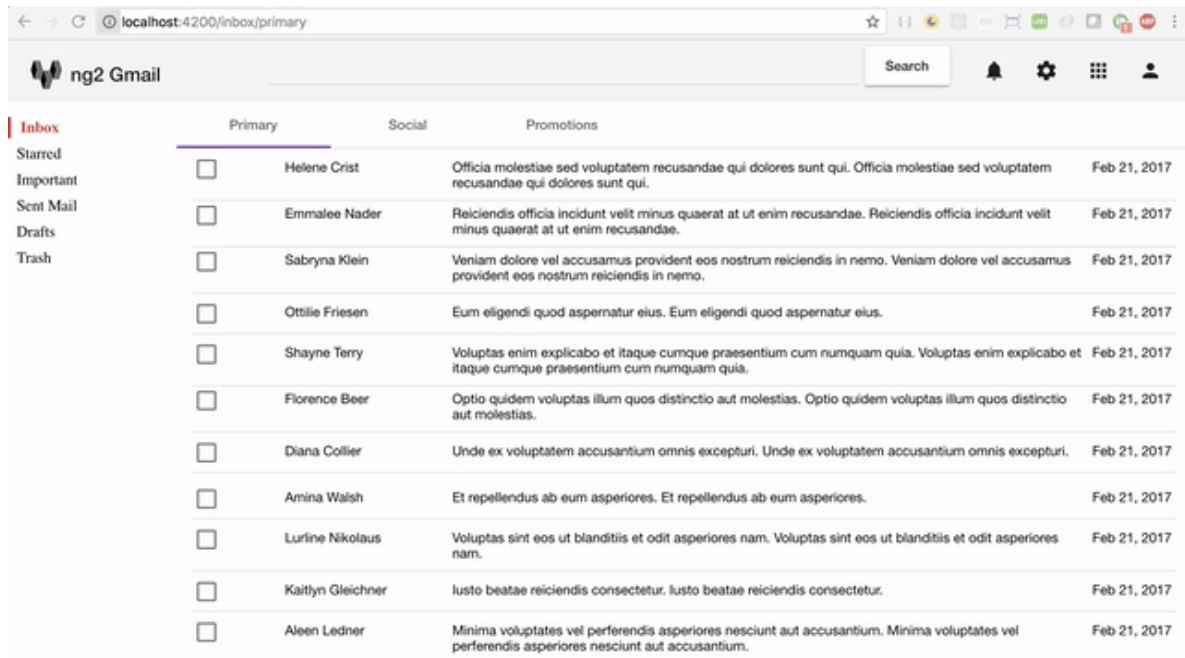
### Lazy Loading routes cải thiện hiệu năng của ứng dụng Angular như thế nào?

Một ứng dụng Angular quy mô lớn sẽ chứa rất nhiều feature modules, nếu chúng được load cùng một lúc khi ứng dụng được khởi động thì sẽ phải mất rất nhiều thời gian. Hãy thử tưởng tượng, người dùng vào website của các bạn mà phải ngồi nhìn icon loading cứ quay vòng vòng đến hàng chục giây thì liệu họ có còn muốn vào lần thứ 2 không ). Các feature modules đó có thể được load bất đồng bộ sau khi ứng dụng được load theo yêu cầu hoặc sử dụng các chiến lược khác nhau. Giảm kích thước của bundle khi ứng dụng được load lần đầu sẽ cải thiện được thời gian load của ứng dụng, do đó sẽ nâng cao trải nghiệm của người dùng đối với ứng dụng web của bạn.

Lazy Loading có nhiều lợi ích:

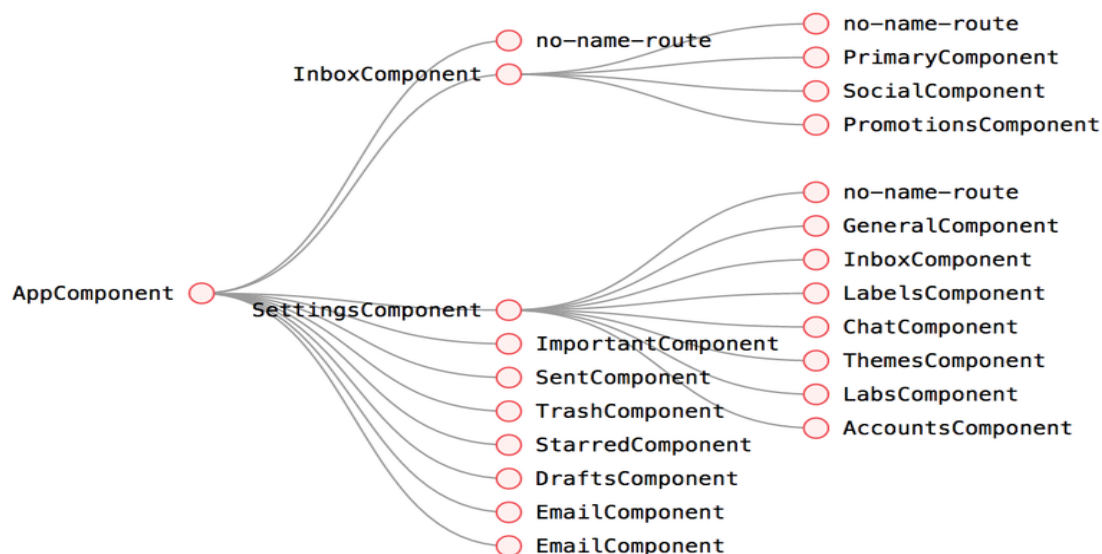
- Bạn có thể load các feature modules chỉ khi được yêu cầu bởi người dùng
- Bạn có thể tăng tốc thời gian load cho người dùng chỉ ghé thăm một số trang nhất định của ứng dụng
- Bạn có thể tiếp tục mở rộng các modules được lazy load mà không tăng kích thước của bundle load lần đầu

Để cho các bạn có cái nhìn chi tiết hơn, ở đây tôi xây dựng một clone đơn giản của Gmail sử dụng những dữ liệu ngẫu nhiên. Ban đầu, code sẽ chưa được lazy load mà sử dụng eager load các feature modules để có thể làm tăng tổng kích thước bundle load ban đầu phục vụ cho việc giải thích ý tưởng của bài viết.



Đây là một ứng dụng đơn giản, nó có một file routing chung cho toàn bộ ứng dụng `app.routing.ts` và tất cả các features được chia ra thành các modules. Tôi đã sử dụng `Augury` extension cho việc debug ứng dụng Angular để có thể inspect được routes của ứng dụng. Cài đặt nó ở đây [augury.angular.io](http://augury.angular.io).

Sau khi cài đặt xong Augury, mở dev tools, chuyển sang tab Augury, sau đó là Router Tree. Dưới đây là router tree cho ứng dụng này:





Ở đây bạn có thể thấy rằng, chúng ta có 2 tầng routing lồng nhau, và mỗi một tầng sẽ có nhiều routes. Trên các ứng dụng thực tế, chúng có thể có nhiều hơn thế và mỗi module sẽ chứa rất nhiều components, templates,... Hiện tại, tất cả các Components trên đều được load cùng một lúc khi mà người dùng mở trang web của bạn. Điều đó là không cần thiết, và nó tốn quá nhiều thời gian chờ đợi của người dùng. Hãy cùng tôi refactor lại codes giúp cho ứng dụng load nhanh hơn nhé ^\_^

### **Refactoring ứng dụng sử dụng lazy load routes**

Tôi sẽ tập trung vào lazy load Settings module và các routes liên quan.

Hiện tại tất cả các routes đều được đăng ký ở trong file `app.routing.ts` và sau đó sẽ được import vào trong root module `app.module.ts`. Dưới đây là file routing hiện tại của ứng dụng:

```
app.routing.ts
const routes: Routes = [
 {
 path: "",
 redirectTo: '/inbox/primary',
 pathMatch: 'full'
 },
 {
 path: 'inbox',
 component: InboxComponent,
 children: [
 {
 path: "",
 redirectTo: 'primary',
 pathMatch: 'full'
 },
 {
 path: 'primary',
 component: PrimaryComponent
 },
]
 }
]
```

```
{
 path: 'social',
 component: SocialComponent
}
...
],
{
 path: 'settings',
 component: SettingsComponent,
 children: [
 {
 path: '',
 redirectTo: 'general',
 pathMatch: 'full'
 },
 {
 path: 'general',
 component: GeneralComponent
 },
 {
 path: 'inbox',
 component: SettingsInboxComponent
 },
 ...
]
},
{
 path: 'important',
 component: ImportantComponent
},
```

```

{
 path: 'sent',
 component: SentComponent
},
...
];

```

**Bước 1:**

Di chuyển routing cho Settings vào trong module của nó `gm-settings.module.ts`. Khi đăng ký các routes với `RouterModule`, chúng ta cần sử dụng `forChild` thay vì `forRoot` giống như đây là routing con của ứng dụng. Do đó, `gm-settings.module.ts` sẽ trông như sau:

```

export const routes: Routes = [
 {
 path: "",
 component: SettingsComponent,
 children: [
 {
 path: "",
 redirectTo: 'general',
 pathMatch: 'full'
 },
 {
 path: 'general',
 component: GeneralComponent
 },
 {
 path: 'inbox',
 component: InboxComponent
 },
 ...
]
 }
];

```

```

]
 }
];
@NgModule({
 imports: [
 CommonModule,
 RouterModule.forChild(routes),
 MdTabsModule
],
 declarations: [
 SettingsComponent,
 GeneralComponent,
 InboxComponent,
 ...
]
})
export class GmSettingsModule { }

```

**Bước 2:**

Cập nhật `app.routing.ts` để load các routes con sử dụng một đường dẫn tương đối đến module `Settings` và thêm cả tên lớp module đó nữa:

```

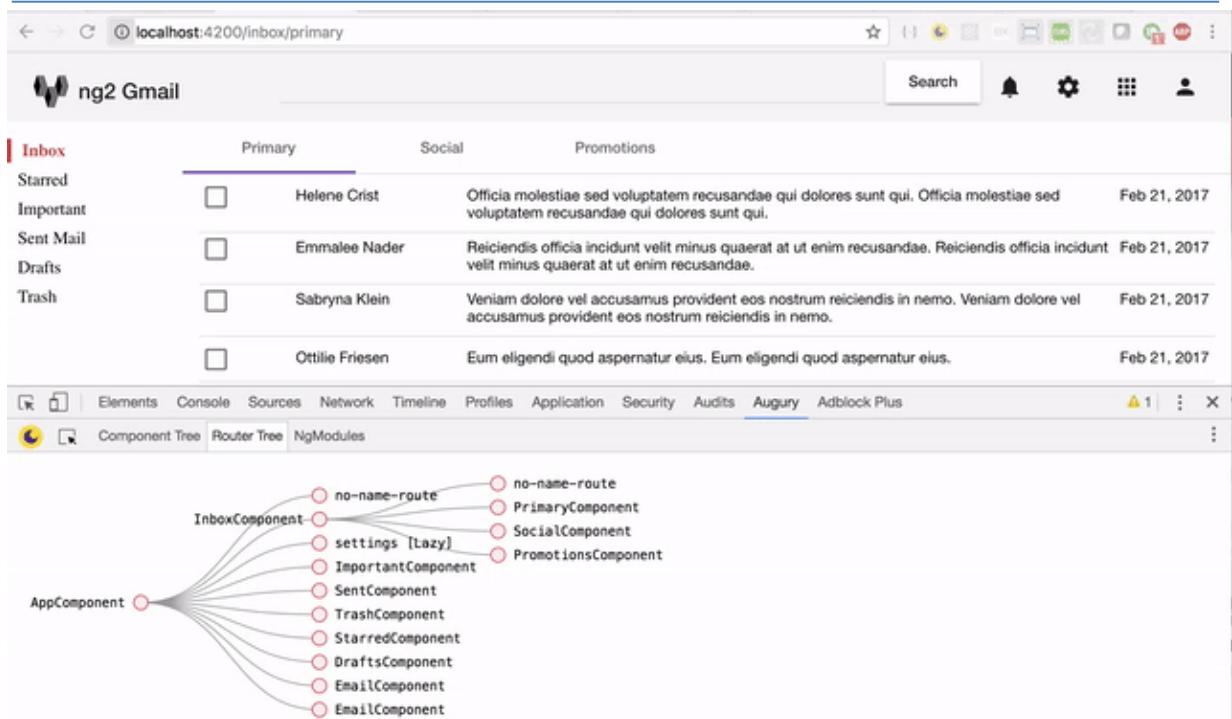
{
 path: 'settings',
 loadChildren: './gm-settings/gm-settings.module#GmSettingsModule'
}

```

**Bước 3:**

Module `Settings` hiện tại đang được import vào trong file `app.module.ts` để được eager load, bây giờ chúng ta cần xóa bỏ nó đi, Angular sẽ tự động cấu hình việc đăng ký module `Settings`.

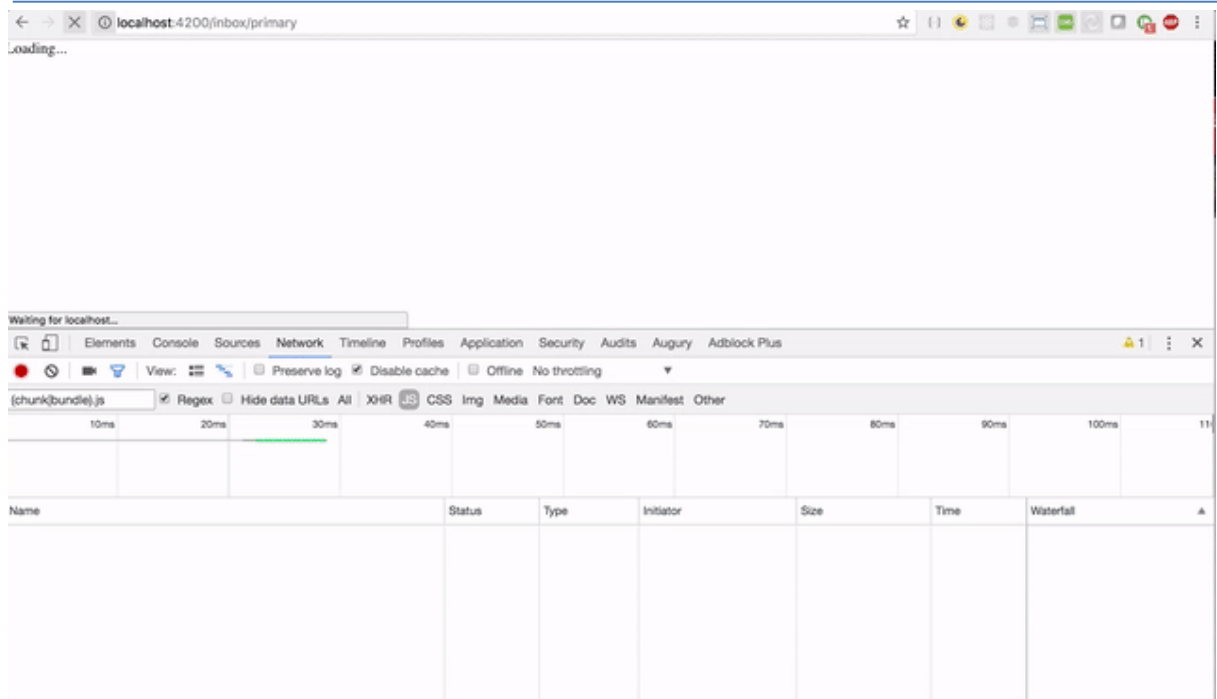
Như vậy là chúng ta đã tiến hành xong việc Lazy Load cho module `Settings`. Hãy cũng xem Router Tree để thấy kết quả thay đổi ra sao nhé:



Nhìn vào kết quả ở trên, ta có thể thấy rằng, khi ứng dụng được load, tất cả các components của module Settings đều chưa hề được load, mà chúng chỉ thực sự được load khi mà người dùng click vào tab settings.

Như vậy là chúng ta đã Lazy Load thành công được module Settings và cải thiện được đáng kể tốc độ load lần đầu của ứng dụng.

Tuy nhiên, có một điều vẫn chưa được tốt cho lắm, đó là các module chỉ được load khi mà người dùng click vào settings hoặc các email cụ thể. Điều này có thể dẫn đến sự chậm trễ khi mà người dùng phải chờ đợi cho module đó được load => làm cho trải nghiệm người dùng xấu đi.

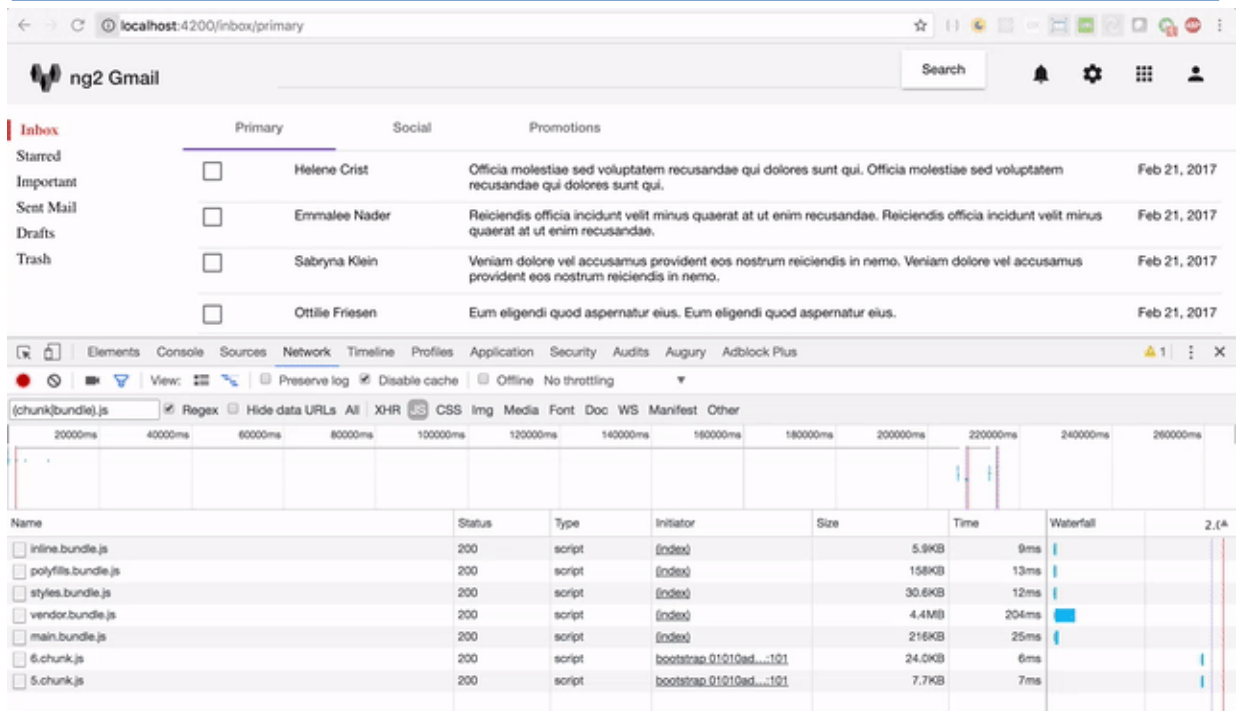


Để khắc phục được vấn đề này, chúng ta cùng tiếp tục phần tiếp theo của bài viết này nhé.

### Chiến lược Preload

Đối với vấn đề được nêu ở trên, Angular cung cấp một phương thức để nói với router load tất cả các lazy load modules bất đồng bộ ngay lập tức sau khi ứng dụng được load và không cần phải chờ cho đến khi chúng được kích hoạt bằng cách người dùng click.

```
app.routing.ts
imports: [
 RouterModule.forRoot(
 routes,
 {
 preloadingStrategy: PreloadAllModules
 }
)
],
```



Nhìn ảnh gif trên ta có thể thấy rằng, tất cả các file \*.chunk.js đều được load ngay lập tức sau khi ứng dụng được khởi tạo.

### Tùy chỉnh chiến lược tải

Sử dụng chiến lược Preload ở trên giúp chúng ta giải quyết được 2 vấn đề:

- Nó sẽ cải thiện tốc độ load của ứng dụng bằng cách giảm kích thước của lần load đầu tiên
- Tất cả các lazy load modules sẽ được load bất đồng bộ ngay sau khi ứng dụng được load xong, do vậy sẽ không có một delay nào đối với người dùng khi chuyển hướng sang bất cứ lazy load module nào.

Tuy nhiên, preload tất cả các lazy load module không phải lúc nào cũng là sự lựa chọn đúng đắn. Đặc biệt đối với các thiết bị di động hay những kết nối băng thông thấp. Chúng ta có thể sẽ phải tải những modules mà người dùng có thể rất ít khi chuyển hướng đến. Tìm ra sự cân bằng cả về hiệu năng và trải nghiệm người dùng là chìa khóa cho việc phát triển.

Ví dụ, trong ứng dụng này, chúng ta có 2 lazy load modules:

- Settings module
- Email module

Đây là một ứng dụng email client nên module Email sẽ được sử dụng rất rất thường xuyên. Tuy nhiên module Settings sẽ được người dùng sử dụng nhưng với tần suất rất

thấp. Do vậy mà việc preload module Email sẽ đem lại hiệu quả cao, trong khi với module Setings thì thấp.

Angular cung cấp một cách extend `PreloadingStrategy` để xác định một tùy chỉnh chiến lược Preload chỉ ra điều kiện cho việc preload các lazy load module. Chúng ta sẽ tạo một provider extend từ `PreloadingStrategy` để preload các modules mà có `preload: true` được xác định trong cấu hình route.

```
custom-preloading.ts
import 'rxjs/add/observable/of';
import { Injectable } from '@angular/core';
import { PreloadingStrategy, Route } from '@angular/router';
import { Observable } from 'rxjs/Observable';

@Injectable()
export class CustomPreloadingStrategy implements PreloadingStrategy {
 preloadedModules: string[] = [];

 preload(route: Route, load: () => Observable<any>): Observable<any> {
 if (route.data && route.data['preload']) {
 this.preloadedModules.push(route.path);
 return load();
 } else {
 return Observable.of(null);
 }
 }
}
```

`CustomPreloadingStrategy` nên được đăng ký vào providers trong module mà `RouterModule.forRoot` được khai báo.

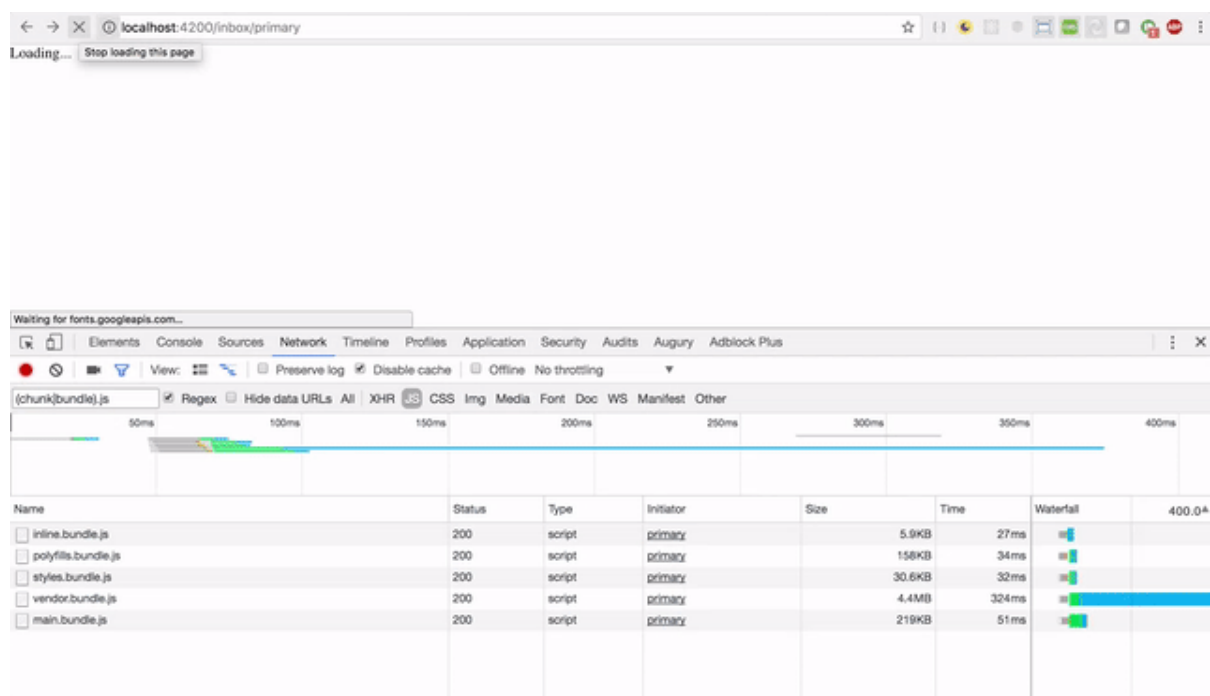
```
app.routing.ts
@NgModule({
 imports: [RouterModule.forRoot(routes, { preloadingStrategy:
 CustomPreloadingStrategy })],
 exports: [RouterModule],
})
```



```
providers: [CustomPreloadingStrategy]
}))
export class AppRoutingModuleModule { }
```

Cuối cùng, trong `app.routing.ts`, chúng ta thêm `data: { preload: true }` vào trong phần khai báo route của module muốn được custom preload:

```
{
 path: ':section',
 loadChildren: './gm-email/gm-email.module#GmEmailModule',
 data: { preload: true }
}
```



Nhìn vào tab network, ta thấy `5.chunks.js` (module Email) đã được preload, còn `6.chunks.js` được load bất động bộ khi người dùng chuyển hướng (module Settings).

Bài viết trên đây đã giới thiệu tổng quát về lazy loading sử dụng trong Angular kết hợp với các chiến lược tải có điều kiện có thể cải thiện hiệu năng của ứng dụng và nâng cao trải nghiệm cho người dùng. Hi vọng bạn đọc sẽ áp dụng nó hiệu quả vào website của mình nhé ^\_^.

## **Bài 7: Thảo luận thiết kế và cài đặt website giới thiệu và đặt hàng online(buổi 1)**

### **Nội dung thảo luận:**

- Thiết kế tổng thể bài toán
- Thiết kế cụ thể bằng Angular một số chức năng:
  - Trang chủ,
  - Trang danh sách sản phẩm,
  - Chi tiết sản phẩm

## Bài 8: Xử lý Form

### 8.1 Template Driven Form

Template-driven forms là phương pháp mà chúng ta sẽ tạo forms dựa vào template (giống như trong Angularjs). Chúng ta thực hiện việc thêm các directives và hành vi vào template, sau đó Angular sẽ tự động tạo forms để quản lý và sử dụng.

#### 8.1.1 Template

Giả sử chúng ta có template form như sau:

```
<form novalidate (submit)="onSubmit()" class="row justify-content-md-center">
 <div class="col-md-8">
 <div class="form-group row">
 <label for="example-text-input" class="col-md-2 col-form-label">Name:</label>
 <div class="col-md-10">
 <input class="form-control" type="text" id="example-text-input">
 </div>
 </div>
 <div class="form-group row">
 <label for="example-email-input" class="col-md-2 col-form-label">Email:</label>
 <div class="col-md-10">
 <input class="form-control" type="email" id="example-email-input">
 </div>
 </div>
 <div class="form-group row">
 <label for="example-url-fb" class="col-md-2 col-form-label">Facebook:</label>
 <div class="col-md-10">
 <input class="form-control" type="url" id="example-url-fb">
 </div>
 </div>
 </div>
</form>
```

```
<label for="example-url-twt" class="col-md-2 col-form-label">Twitter:</label>
<div class="col-md-10">
 <input class="form-control" type="url" id="example-url-twt">
</div>
</div>
<div class="form-group row">
 <label for="example-url-web" class="col-md-2 col-form-label">Website:</label>
 <div class="col-md-10">
 <input class="form-control" type="url" id="example-url-web">
 </div>
</div>
<div class="form-group row">
 <label for="example-tel-input" class="col-md-2 col-form-label">Tel:</label>
 <div class="col-md-10">
 <input class="form-control" type="tel" id="example-tel-input">
 </div>
</div>
<div class="form-group row">
 <div class="col-md-10 offset-md-2">
 <button class="btn btn-primary" type="submit">Submit</button>
 </div>
</div>
</div>
</form>
```

Trên đây chỉ là một form HTML thông thường, khi browser render chúng ta sẽ có form trông giống như sau:

## Angular Template-Driven Forms

Name:

Email:

Facebook:

Twitter:

Website:

Tel:

### 8.1.2 Import APIs cho Template-driven forms

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

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

@NgModule({
 declarations: [...],
 imports: [
 ...,
 FormsModule
],
 providers: [...],
 bootstrap: [...]
})

export class AppModule { }
```

### 8.1.3 ngForm và ngModel directives

Nhiệm vụ đầu tiên chúng ta cần làm là truy cập vào form instance và gán các control vào form. Chúng ta sẽ lần lượt sử dụng `ngForm` và `ngModel` directives như sau:

```
<form novalidate #form="ngForm" ...>
</form>
```

Angular cung cấp một giải pháp để có thể truy cập được directive/component instance ở trong template của component bằng cách sử dụng `exportAs` trong khai báo directive/component metadata.

Ở trong đoạn code phía trên, chúng ta đã tạo một template variable là `form`, nó sẽ là một instance của directive `ngForm`, như thế chúng ta có thể sử dụng các public API mà directive này cung cấp như lấy ra value của nó chẳng hạn như `form.value`.

Giờ đây chúng ta có thể sử dụng form value cho việc submit form chẳng hạn.

```
<form novalidate #form="ngForm"
(submit)="onSubmit(form.value)" ...>
</form>
<p>Form value:</p>
<pre>{{ form.value | json }}</pre>
```

Và để dễ dàng trong quá trình development, chúng ta có thể thêm một phần hiển thị ở template để biết được form value đang có gì như hai dòng code cuối ở trên.

Tại thời điểm này, cho dù chúng ta có form control ở template nhưng Angular không thể biết cái nào cần quản lý nên chúng ta chỉ nhận được một object rỗng.

Bây giờ công việc tiếp theo là chúng ta phải nói cho Angular biết các form control nào cần phải quản lý. Đây chính là lúc chúng ta dùng đến `ngModel` directive.

Chúng ta sẽ thêm `ngModel` vào các control như sau:

```
<input class="form-control" type="text" ngModel ...>
```

Nhưng nếu bạn không khai báo attribute name cho form control, bạn sẽ gặp phải một lỗi giống như sau:

*Error: If ngModel is used within a form tag, either the name attribute must be set or the form control must be defined as 'standalone' in ngModelOptions.*

Kèm với đó là bạn sẽ có các ví dụ để sửa lỗi trên.

Chúng ta cần thêm một số config để Angular biết cách tạo ra form control của nó để quản lý. Và chúng ta sẽ thêm attribute `name` cho các form control ở template trên.

```
<input class="form-control" type="text" ngModel name="contact-name" ...>
```

Bây giờ quan sát form value chúng ta sẽ có một object có key `contact-name` chẳng hạn:

```
{
 "contact-name": ""
}
```

Nếu bạn quen với camel case, chúng ta có thể sửa đổi chút để object của chúng ta có key nhìn quen thuộc hơn chẳng hạn.

```
<input class="form-control" type="text" ngModel name="contactName" ...>
```

Kết quả nhận được:

```
{
 "contactName": ""
}
```

Giờ chúng ta có thể cài đặt tương tự cho các phần tử khác của form.

Và khi bạn nhập giá trị cho các control thì Angular sẽ tự cập nhật cho các control của form tương ứng, chẳng hạn sau khi nhập xong và submit thì form sẽ có value như sau:

```
{
 "contactName": "Tiep Phan",
 "email": "abc@deg.com",
 "facebook": "facebook.com",
 "twitter": "twitter.com",
 "website": "tiepphan.com",
 "tel": "1234-5678-90"
}
```

Bây giờ có một tình huống phát sinh là bạn cần bind data cho các control với một dữ liệu có sẵn, lúc này chúng ta sẽ dùng đến binding cho property, và property chúng ta nhắc đến ở đây chính là `ngModel`.

Chúng ta có dạng binding quen thuộc như sau:

```
[ngModel]="obj.prop"
```

Giả sử object mà chúng ta có ở đây có dạng:

```
contact = {
 "contactName": "Tiep Phan",
 ...
}
```

```

 "email": "abc@deg.com",
 "facebook": "facebook.com",
 "twitter": "twitter.com",
 "website": "tiepphan.com",
 "tel": "1234-5678-90"
 }

```

Template của chúng ta sẽ thay đổi như sau:

```

<input [ngModel]="contact.contactName" name="contactName" class="form-control" type="text" ...>

```

Mọi thứ đều bắt nguồn từ những điều cơ bản nhất, `[ngModel]` chính là one-way binding mà chúng ta vẫn thường dùng.

Lưu ý rằng, khi bạn update form control, bản thân control được form quản lý sẽ thay đổi – `form.value`, nhưng object contact ở trên sẽ không hề hấn gì, vì chúng ta không hề đụng chạm gì tới nó, chúng ta chỉ binding một chiều, mà không binding ngược trở lại. Điều này dẫn đến chúng ta có thêm một dạng khác của `ngModel` đó là cú pháp two-way binding `[(ngModel)]`.

```

<input [(ngModel)]="contact.contactName" name="contactName" class="form-control" type="text" ...>

```

Như vậy, nếu bạn không cần binding thì chỉ cần thêm `ngModel` là đủ, nếu bạn muốn one-way binding thì sử dụng `[ngModel]`, cuối cùng là muốn dùng two-way binding với `[(ngModel)]`. Mặc dù vậy, khi bạn thay đổi form control value, thì Angular sẽ cập nhật lại giá trị của các control của form mà nó đang quản lý.

#### 8.1.4 ngModelGroup directive

Đến lúc này, chúng ta vẫn đang chỉ quản lý form control với một object chứa tất cả các keys cần thiết, vậy làm thế nào chúng ta có thể gom nhóm một số key lại thành một group riêng, câu trả lời là `ngModelGroup`. Directive này tạo ra một group lồng vào group cha, giống như object nằm trong một object khác.

Giả sử như template kể trên, chúng ta sẽ nhóm các url thành một nhóm có tên là `social` chẳng hạn:

```

<fieldset ngModelGroup="social">
 <div class="form-group row">

```



```

<label for="example-url-fb" class="col-md-2 col-form-label">
 Facebook:
</label>
<div class="col-md-10">
 <input class="form-control" type="url" id="example-url-fb"
 ngModel name="facebook">
</div>
</div>
<div class="form-group row">
 <label for="example-url-twt" class="col-md-2 col-form-label">
 Twitter:
 </label>
 <div class="col-md-10">
 <input class="form-control" type="url" id="example-url-twt"
 ngModel name="twitter">
 </div>
</div>
<div class="form-group row">
 <label for="example-url-web" class="col-md-2 col-form-label">
 Website:
 </label>
 <div class="col-md-10">
 <input class="form-control" type="url" id="example-url-web"
 ngModel name="website">
 </div>
</div>
</fieldset>

```

Kết quả thu được chúng ta có form value với cấu trúc:

```

{
 "contactName": "",
 "email": "",
 "social": {

```

```

 "facebook": "",
 "twitter": "",
 "website": ""
 },
 "tel": ""
}

```

### 8.1.5 Submit form

Lưu ý rằng phần nội dung này có thể áp dụng cho cả Reactive forms mà chúng ta sẽ tìm hiểu tiếp theo.

Ở phần trước, chúng ta đã listen event `submit` của form, nhưng ngoài ra, còn một event khác cũng được fired ra khi thực hiện submit form, đó là `ngSubmit`. Vậy có điều gì khác biệt giữa `submit` và `ngSubmit`?

Giống như `submit`, event `ngSubmit` cũng thực hiện hành động khi form thực hiện submit – người dùng nhấn vào button submit chẳng hạn. Nhưng `ngSubmit` sẽ thêm một số nhiệm vụ để đảm bảo form của bạn không thực hiện submit form theo cách thông thường – tải lại trang sau khi submit.

Giả sử, chúng ta thực hiện một tác vụ nào đó trong hàm listen form submit mà sinh ra exception, lúc này nếu bạn sử dụng `submit`, trang web của bạn sẽ reload, còn nếu bạn sử dụng `ngSubmit`, nó sẽ không reload – phiên bản lúc này tôi đang sử dụng.

```

onSubmit(formValue) {
 // Do something awesome
 console.log(formValue);
 throw Error('something go wrong');
}

```

Lời khuyên dành cho bạn là nên dùng `ngSubmit` cho việc listen form submit.

### 8.1.6 Template-driven error validation

Có vẻ như user nhập sai thông tin rồi, làm sao tôi có thể hiển thị thông báo cho họ biết để sửa đây?

Angular đã có sẵn tính năng cơ bản cho việc validation của bạn. Bây giờ hãy bắt đầu với việc kiểm tra lỗi và cảnh báo.

Angular cung cấp một số Validators cơ bản mà bạn có thể dùng ngay trong template như: `required`, `minlength`, `maxlength`, `pattern` và từ Angular v4 trở đi có thêm `email`.

Chúng được viết là các directives, nên bạn có thể sử dụng như các directives khác trong template của bạn.

Chúng ta sẽ bỏ qua validation của HTML5, vậy nên ngay từ đầu form chúng ta đã thêm novalidate attribute vào khai báo form, thay vào đó là sử dụng Angular validation.

Giả sử chúng ta cần cài đặt contactName là required, chúng ta sẽ đặt như sau:

```
<input ngModel name="contactName" required class="form-control" type="text" ...>
```

Và để dễ dàng quan sát, chúng ta sẽ thêm phản hiển thị lỗi như sau:

```
<p>Form contactName errors:</p>
```

```
<pre>{{ form.controls.contactName?.errors | json }}</pre>
```

Chúng ta sử dụng **safe navigation operator** để truy cập property của một object có thể bị null/undefined mà không gây ra lỗi chương trình.

Khi không nhập gì vào input contactName, chúng ta có thể thấy một thông báo lỗi như sau:

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

Khi input này được nhập dữ liệu thì chúng ta sẽ thấy key required của object trên sẽ bị xóa bỏ.

Công việc của chúng ta bây giờ là sử dụng ngIf chẳng hạn để show/hide error cho người dùng được biết.

```
<div class="col alert alert-danger" role="alert"
 *ngIf="form.controls.contactName?.errors?.required">
 Name is required!
</div>
```

Và có thể thêm việc không cho người dùng nhấn button submit khi trạng thái của form là invalid như sau:

```
<button class="btn btn-primary" type="submit"
 [disabled]="form.invalid">
 Submit
</button>
```

Bây giờ, nếu bạn muốn chỉ hiển thị thông báo error khi người dùng đã focus vào input đó mà không nhập gì, lúc này chúng ta có thể thông qua trạng thái của form bằng cách truy cập các properties như touched, dirty hay pristine, ...

Trong đó:

- touched: true nếu người dùng đã focus vào input rồi không focus vào nữa.
- untouched: true nếu người dùng chưa đụng chạm gì hoặc lần đầu tiên focus và chưa bị mất focus (ngược lại với touched)
- dirty: true nếu người dùng đã tương tác với control – nhập một ký tự vào input text chẳng hạn.
- pristine: true nếu người dùng chưa tương tác gì với control, mặc dù có thể đã touched, nhưng chưa sửa đổi gì.

Chúng ta sẽ thay đổi một chút form validation:

```
<div class="col alert alert-danger" role="alert"
 *ngIf="form.controls.contactName?.errors?.required &&
 form.controls.contactName?.touched">
 Name is required!
</div>
```

Giờ đây chỉ khi nào người dùng touched vào control và có error thì validation message sẽ hiển thị.

Chúng ta muốn dùng template variable để truy cập control thay vì truy cập dài dài như trên có được không?

Câu trả lời là có, chúng ta hoàn toàn có thể dùng template variable như sau:

```
<input ngModel name="contactName" required #contactName="ngModel"
 class="form-control" type="text" ...>
```

Và sử dụng như một template variable thông thường:

```
<div class="col alert alert-danger" role="alert"
 *ngIf="contactName?.errors?.required && contactName?.touched">
 Name is required!
</div>
```

Như vậy, việc sử dụng Template-driven form trong Angular khá dễ dàng, trong phần này chúng ta chưa đề cập đến custom validation cho form control. Thay vào đó chúng ta sẽ có một phần riêng để thảo luận về tính năng này.

## 8.2 Model Driven Form(Reactive Form)

Thuật ngữ **Reactive Forms** hay còn được gọi là **Model-Driven Forms**, là một phương pháp để tạo form trong Angular, phương pháp này tránh việc sử dụng các directive ví dụ như ngModel, required, etc, thay vào đó tạo các Object Model ở trong các Component, rồi tạo ra form từ chúng.

Một điều lưu ý đó là Template-Driven là async còn Reactive là sync.

Trong Reactive forms, chúng ta tạo toàn bộ form control tree 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.

Trong Template-driven forms, chúng ta ủy thác việc tạo form control cho directives, để tránh bị lỗi `changed after checked`, directives cần một cycle nữa để build toàn bộ form control tree. Vậy nên bạn cần đợi một tick nữa để có thể truy cập vào các phần tử của form. Chính điều này khiến việc test template-driven form trở nên phức tạp hơn.

Bạn có thể tham khảo thêm tại document sau:

<https://angular.io/docs/ts/latest/guide/reactive-forms.html#!#async-vs-sync>

### 8.2.1 Các thành phần cơ bản của form

- AbstractControl là một abstract class cho 3 lớp con form control: FormControl, FormGroup, và FormArray. Nó cung cấp các hành vi, thuộc tính chung cho các lớp con.
- 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.
- 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 – đây là một dạng composite. Ở level cao nhất của một form trong component của bạn là một FormGroup.
- FormArray nó lưu giữ giá trị và trạng thái hợp lệ của một mảng các đối tượng thuộc AbstractControl giống như FormGroup. Nó cũng là một dạng composite. Nhưng nó không phải là thành phần ở level cao nhất.

### 8.2.2 Template

Cũng giống như trong [Template-driven forms](#), chúng ta sẽ sử dụng một template giống thế.

```
<form novalidate (ngSubmit)="onSubmit()"
 class="row justify-content-md-center">
 <div class="col-md-8">
 <div class="form-group row">
 <label for="example-text-input" class="col-2 col-form-label">
 Name:
 </label>
 <div class="col-10">
 <input class="form-control" type="text"
 id="example-text-input">
 </div>
 </div>
 <div class="form-group row">
 <label for="example-email-input" class="col-2 col-form-label">
 Email:
 </label>
 <div class="col-10">
 <input class="form-control" type="email"
 id="example-email-input">
 </div>
 </div>
 <div class="form-group row">
 <label for="example-url-fb" class="col-2 col-form-label">
 Facebook:
 </label>
 <div class="col-10">
 <input class="form-control" type="url" id="example-url-fb">
 </div>
 </div>
 </div>
```

```
<div class="form-group row">
 <label for="example-url-twt" class="col-2 col-form-label">
 Twitter:
 </label>
 <div class="col-10">
 <input class="form-control" type="url" id="example-url-twt">
 </div>
</div>

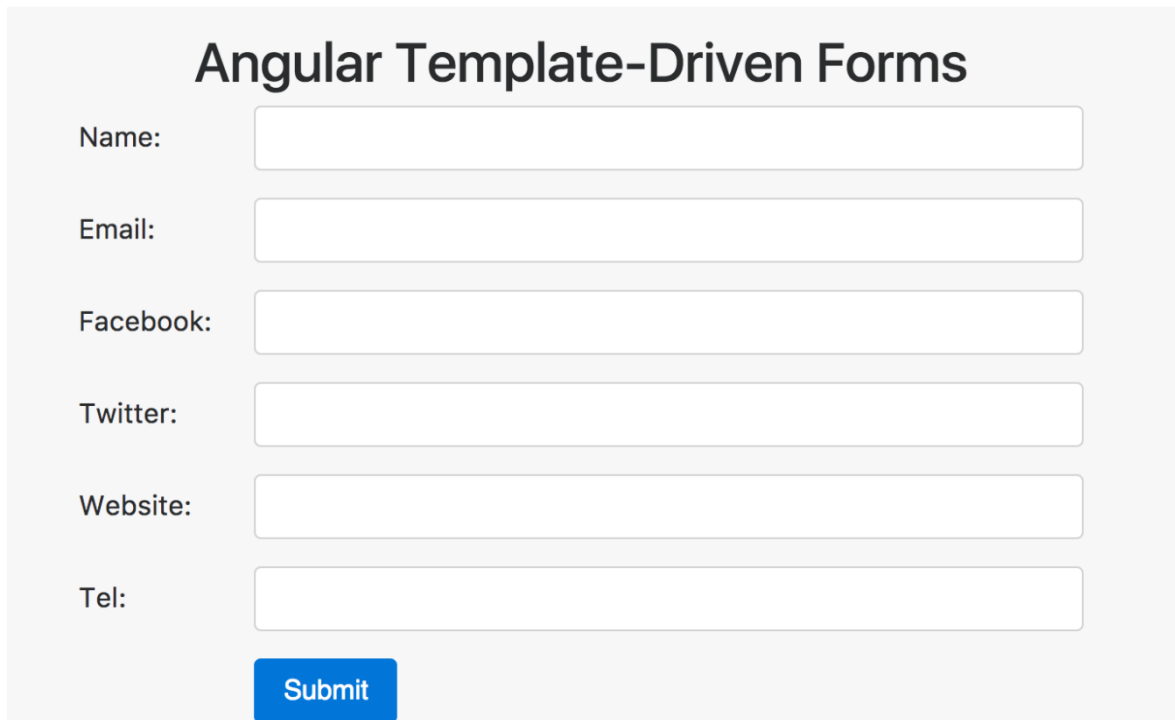
<div class="form-group row">
 <label for="example-url-web" class="col-2 col-form-label">
 Website:
 </label>
 <div class="col-10">
 <input class="form-control" type="url" id="example-url-web">
 </div>
</div>

<div class="form-group row">
 <label for="example-tel-input" class="col-2 col-form-label">
 Tel:
 </label>
 <div class="col-10">
 <input class="form-control" type="tel"
 id="example-tel-input">
 </div>
</div>

<div class="form-group row">
 <div class="col-10 offset-2">
 <button class="btn btn-primary"
 type="submit">Submit</button>
 </div>
</div>
</div>
```

&lt;/form&gt;

Trên đây chỉ là một form HTML thông thường, khi browser render chúng ta sẽ có form trông giống như sau:



**Angular Template-Driven Forms**

Name:

Email:

Facebook:

Twitter:

Website:

Tel:

Nhưng trong template trên chúng ta mặc định sử dụng `ngSubmit`, các bạn có thể trở lại bài trước để hiểu về `ngSubmit` vs `submit` event.

### 8.2.3 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:

Lưu ý rằng bạn có thể import cả Reactive form và Template-driven form modules.

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

```
@NgModule({
 declarations: [...],
 imports: [
 ...,
 ReactiveFormsModule
],
})
```



```

providers: [...],
bootstrap: [...]
})
export class AppModule { }

```

#### 8.2.4 Khởi tạo form trong Component

Như đã nói ở phần trước, chúng ta có thể khởi tạo form trong Component ở một trong 3 giai đoạn: khởi tạo ngay lúc khai báo property, trong constructor, trong ngOnInit. Để thống nhất về code, chúng ta sẽ khởi tạo trong ngOnInit.

Giả sử chúng ta tạo một component như sau:

```

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

@Component({
 selector: 'tpc-contact-reactive-form',
 templateUrl: './contact-reactive-form.component.html',
 styleUrls: ['./contact-reactive-form.component.scss']
})
export class ContactReactiveFormComponent implements OnInit {
 rfContact: FormGroup;
 constructor() { }
 ngOnInit() {
 this.rfContact = new FormGroup({
 contactName: new FormControl(),
 email: new FormControl(),
 social: new FormGroup({
 facebook: new FormControl(),
 twitter: new FormControl(),
 website: new FormControl()
 }),
 tel: new FormControl()
 });
 }
 onSubmit() {

```

```
// Do something awesome
console.log(this.rfContact);
}
}
```

### 8.2.5 Binding controls

Ở top-level (form), chúng ta sẽ bind một formGroup như sau:

```
<form novalidate (ngSubmit)="onSubmit()" [formGroup]="rfContact"
 class="row justify-content-md-center">
 </form>
```

Lần lượt với các FormControl instances, chúng ta sẽ bind với property `formControlName` như sau:

```
<input class="form-control" type="text" id="example-text-input"
 formControlName="contactName">
```

Đối với formGroup lồng trong formGroup như cấu trúc ở trên, chúng ta có thêm một property cần bind là `formGroupName`:

```
<fieldset formGroupName="social">
 <div class="form-group row">
 <label for="example-url-fb" class="col-md-2 col-form-label">Facebook:</label>
 <div class="col-md-10">
 <input class="form-control" type="url" id="example-url-fb"
 formControlName="facebook">
 </div>
 </div>
 <div class="form-group row">
 <label for="example-url-twt" class="col-md-2 col-form-label">Twitter:</label>
 <div class="col-md-10">
 <input class="form-control" type="url" id="example-url-twt"
 formControlName="twitter">
 </div>
 </div>
```

```

<div class="form-group row">
 <label for="example-url-web" class="col-md-2 col-form-
label">Website:</label>
 <div class="col-md-10">
 <input class="form-control" type="url" id="example-url-web"
formControlName="website">
 </div>
</div>
</fieldset>

```

Việc binding property vào template cần map cấu trúc cây DOM giống với cấu trúc của model ở component, bạn không thể binding một property không thuộc về một control, chẳng hạn ở model trên, control facebook thuộc về group social, nên bạn không thể bind control này trực tiếp vào group rfContact.

### 8.2.6 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:

Lưu ý: có 2 loại validators, một là validator validator đồng bộ (sync), ví dụ required, minlength, etc. Hai là async validator (validator bất đồng bộ) chẳng hạn như validator để xem user tồn tại chưa, lúc này bạn gọi AJAX để thực hiện tìm kiếm và đó là một quá trình xử lý bất đồng bộ. Các class FormControl và FormGroup cho phép thêm validator theo thứ tự sync – async vào danh sách tham số của constructor.

```

export declare class FormGroup extends AbstractControl {
 constructor(controls: {
 [key: string]: AbstractControl;
 }, validator?: ValidatorFn | null,
 asyncValidator?: AsyncValidatorFn | null);
}

```

Tương tự cho FormArray.

Và FormControl:

```

export declare class FormControl extends AbstractControl {
 constructor(formState?: any,
 validator?: ValidatorFn | ValidatorFn[] | null,

```

```

 asyncValidator?: AsyncValidatorFn | AsyncValidatorFn[] | null);
 }

```

Giả sử chúng ta thêm required cho form control `contactName` ở trên, chúng ta chỉ cần thêm vào constructor như sau:

```
contactName: new FormControl("", Validators.required)
```

Với `Validators.required` là một validator function.

Hoặc có thể thêm nhiều validator với `Validators.compose`, hoặc truyền vào là một mảng các validator functions:

```
contactName: new FormControl(
 [Validators.required, Validators.minLength(3)]
)
```

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

Để hiển thị thông báo cho người dùng biết về lỗi, chúng ta có thể sử dụng phương pháp như đã đề cập trong phần Template-driven form.

```

<div class="col alert alert-danger" role="alert"
 *ngIf="rfContact.controls.contactName?.errors?.required
 && rfContact.controls.contactName?.touched">
 Name is required!
</div>

```

Hoặc sử dụng `hasError`:

```

<div class="col alert alert-danger" role="alert"
 *ngIf="rfContact.controls.contactName?.hasError('required')
 && rfContact.controls.contactName?.touched">
 Name is required!
</div>

```

Không những thế, Reactive form còn cho phép chúng ta lấy ra một control theo path đến nó một cách dễ dàng, ví dụ để lấy về `contactName`:

```
rfContact.get('contactName')
```

Hoặc với control `facebook`:

```
rfContact.get('social.facebook')
```

// or

```
rfContact.get(['social', 'facebook'])
```

Với phương pháp này chúng ta không cần tạo nhiều template variable nữa.

### **Tạo custom validator**

Trong ví dụ này, ta sẽ tạo 1 custom validator cho trường pwd: độ dài tối thiểu phải là 6 kí tự và phải chứa ký tự đặc biệt

Để tạo custom validator cho form, ta cần khai báo và sử dụng hàm xử lý validator. Ở đây, chúng ta khai báo hàm pwdCheckValidator để kiểm tra độ dài của form control

*app.component.html*

```
<div>
 <form [formGroup]="formdata" (ngSubmit) =
 "onClickSubmit(formdata.value)" >
 <input type="text" class="fortextbox" name="username"
 placeholder="username"
 formControlName="username">

 <input type="pwd" class="fortextbox" name="pwd" type="password"
 placeholder="pwd" formControlName="pwd">

 <input type="submit" [disabled] = "!formdata.valid" class="forsubmit"
 value="Log In">
 </form>
</div>
<p>
 Bạn vừa nhập username : {{username}}
</p>
```

*app.component.ts*

```
import { Component } from '@angular/core';
import { FormGroup, FormControl, Validators } from '@angular/forms';
@Component({
 selector: 'app-root',
 templateUrl: './app.component.html',
 styleUrls: ['./app.component.css']
})
```

```

export class AppComponent {
 username;
 formdata;
 ngOnInit() {
 this.formdata = new FormGroup({
 username: new FormControl("dongnh",
[Validators.required,Validators.minLength(3), Validators.maxLength(5)]),
 pwd: new FormControl("123", [this.pwdCheckValidator])
 });
 }
 onClickSubmit(data) { this.username = data.username; }
 pwdCheckValidator(control){
 var filteredStrings = {search:control.value, select:'@#!$%&*' }
 var result = (filteredStrings.select.match(new RegExp('[' +
filteredStrings.search + ']', 'g')) || []).join("");
 if(control.value.length < 6 && !result){
 return {pwd: true};
 }
 }
}

```

Nếu như độ dài của giá trị form control < 6 và không chứa ký tự đặc biệt thì ta return lại object chứa trạng thái của control bị lỗi hay không. Ở đây chúng ta return lại là return {pwd: true}; có nghĩa là form control pwd đang bị lỗi.

Bây giờ thử chạy ứng dụng trên trình duyệt. Nếu nhập password dưới 6 ký tự và không chứa ký tự đặc biệt thì button submit sẽ bị disable



### 8.2.7 Form Builder

Một trong những tính năng tuyệt vời của Angular form là Form Builder, nó giúp bạn tạo form model một cách nhanh chóng và tiện lợi. Việc bạn phải làm là inject class FormBuilder vào class bạn muốn tạo form, và gọi các API như `group`, `array`, `control` để tạo form.

```
constructor(private fb: FormBuilder) { }
ngOnInit() {
 this.formdata = this.fb.group({
 username: this.fb.control("", [Validators.required, Validators.minLength(3)]),
 pwd: this.fb.control(""),
 });
}
```

Hoặc một cách rút gọn hơn nữa:

```
this.formdata = this.fb.group({
 username: ['', [Validators.required, Validators.minLength(3)]],
 pwd: ''
});
```

### 8.2.8 FormArray

Giả sử bây giờ bạn muốn người dùng có thể nhập vào một hoặc nhiều số điện thoại. Vậy có cách nào tạo form với số lượng thay đổi như thế hay không. Rất may cho chúng ta, Reactive form có một loại control là FormArray, nó sẽ giúp chúng ta làm được việc đó, bây giờ hãy thay đổi code một chút.

```
ngOnInit() {
 this.rfContact = this.fb.group({
 // ...
 tels: this.fb.array([
 this.fb.control("")
])
 });
}

get tels(): FormArray {
 return this.rfContact.get('tels') as FormArray;
}
```

```

}

addTel() {
 this.tels.push(this.fb.control(""));
}

removeTel(index: number) {
 this.tels.removeAt(index);
}

```

Và template của chúng ta sẽ có:

```

<div formArrayName="tels" *ngIf="tels.controls.length">
 <div class="form-group row" *ngFor="let c of tels.controls; index as i">
 <label for="example-tel-input" class="col-md-2 col-form-label">Tel:</label>
 <div class="col-md-10">
 <input class="form-control" type="tel" id="example-tel-input"
[formControlName]="i">
 <div class="text-right">
 <button class="btn btn-info" (click)="addTel()">+</button>
 <button class="btn btn-danger" (click)="removeTel(i)"
*ngIf="tels.controls.length > 1">-</button>
 </div>
 </div>
 </div>
</div>

```

Khi render chúng ta sẽ có form có dạng:

Ở đoạn code trên chúng ta đã tạo ra một `FormArray` instance và khi binding vào template chúng ta thông báo nó với directive `formArrayName`. Khi thực hiện việc lặp, chúng ta tạo ra biến index có tên là `"i"`, với mỗi biến index như thế, Angular sẽ lưu trữ



tương ứng với một phần tử trong FormArray là một FormControl instance, trong trường hợp này của chúng ta là một FormControl instance, vậy nên chúng ta có đoạn binding property như sau: `[formControlName]="i"`.

FormArray cung cấp một số phương thức cho phép chúng ta thêm, xóa phần tử trong array như `insert`, `push`, `removeAt`. Hay phương thức `at` để lấy ra phần tử ở vị trí cụ thể. Chúng sẽ có ích khi bạn sử dụng trong ứng dụng của mình – trong trường hợp ở trên chúng ta có dùng để thêm hoặc xóa.

### 8.2.9 Forms with a single control

Giả sử bạn có một form search chẳng hạn, bạn chỉ có một form control, lúc này bạn không muốn phải bao cả form và form group, có cách nào làm được thế không.

Đây là lúc `formControl` directive phát huy tác dụng. Chúng ta hãy xem xét search form sau đây.

```
<input type="search" class="form-control" [formControl]="searchControl">
```

Còn đây là component code:

```
export class SearchComponent implements OnInit {
 searchControl = new FormControl();
 ngOnInit() {
 this.searchControl.valueChanges.subscribe(value => {
 // do search with value here
 console.log(value);
 });
 }
}
```

Giờ đây, bạn có thể tạo ra một form search cực kỳ nhanh và đơn giản phải không nào.

### 8.2.10 Cập nhật giá trị cho form, control

Có 2 phương thức để cập nhật giá trị cho form control được mô tả bởi class `AbstractControl` là `setValue` và `patchValue`. Chúng là các abstract method, vậy nên các class dẫn xuất sẽ phải implement riêng cho chúng.

Đối với class `FormControl`, không có gì khác biệt giữa 2 phương thức – thực chất `patchValue` gọi lại `setValue`.

Đối với các class `FormGroup` và `FormArray`, `patchValue` sẽ cập nhật các giá trị được khai báo tương ứng trong object value truyền vào. Nhưng `setValue` sẽ báo lỗi nếu một

control nào bị thiếu hoặc thừa, tức là bạn phải truyền chính xác object có cấu trúc giống như cấu trúc của form hay nói cách khác là không chấp nhận subset hoặc superset của cấu trúc form hiện tại.

Vậy nên nếu bạn muốn cập nhật một phần của form thì hãy dùng `patchValue`, nếu bạn muốn set lại tất cả và đảm bảo không cái nào bị thiếu thì dùng `setValue` để tận dụng việc báo lỗi của nó.

Ngoài ra, còn có phương thức `reset` để bạn có thể reset lại trạng thái lúc khởi tạo của form hoặc control.

## Bài 9: Services & Dependency Injection

### 9.1 Service là gì?

Service (dịch vụ) chẳng qua cũng là một cách giúp cho chúng ta tái sử dụng code mà thôi, chẳng hạn như bạn có một lớp `TimeService`, thì thay vì mỗi lần cần lấy các đối tượng `TimeService` đang có, chúng ta phải viết code để tạo đối tượng, truyền tham số...v.v ở nhiều nơi khác nhau, thì bây giờ chúng ta chỉ cần viết một lớp service làm điều đó luôn cho chúng ta, như vậy việc quản lý code sẽ dễ dàng hơn, chẳng hạn như mỗi lần thay đổi phương thức khởi tạo, thì chúng ta chỉ cần thay đổi code trong lớp service là được, thay vì phải đi sửa lại toàn bộ những dòng code khởi tạo đó.

Dependency Injection là chức năng cho phép chúng ta “nhúng” các lớp vào các lớp khác, giống như dùng một thư viện vậy, và chúng ta có thể dùng các lớp được nhúng vào đó giống như dùng một thuộc tính bình thường mà không cần phải thực hiện các công đoạn khai báo, khởi tạo...v.v Tất nhiên là service thì chỉ có code không hề có giao diện.

Để tạo ra một service thì chúng ta cần import và mô tả một class với từ khóa

`@injectable` lấy từ `@angular/core` module.

Ta hãy lấy một ví dụ:

```
import { Injectable } from '@angular/core';

@Injectable()
export class TimeService {
 constructor() { }
 getTime(){
 return `${new Date().getHours()} : ${new Date().getMinutes()} : ${new
 Date().getSeconds()}`;
 }
}
```

Sau khi định nghĩa class `TimeService` là `@Injectable()` thì chúng ta sẽ gọi được service này ở nhiều chỗ component khác trong ứng dụng.

### 9.2 Dependency Injection (DI)

Khi một class muốn được gọi (được tiêm vào, inject vào) một component cần gọi hàm bên trong nó, chúng ta cần dùng đến Dependency Injection. Và rất đơn giản chỉ

cần gọi ở hàm khởi tạo (constructor) của component là sẽ tiêm được service vào để dùng nó bên trong component đó.

```
import { Component } from '@angular/core';
import { TimeService } from './time.service';

@Component({
 selector: 'app-root',
 providers: [TimeService],
 template: `<div>Date: {{timeValue}}</div>`,
})
export class SampleComponent {
 timeValue: string;

 constructor(private time: TimeService){
 this.timeValue = this.time.getTime(); //Gọi biến time từ
 }
}
```

Sau đó chúng ta khai báo phương thức constructor(), phương thức này có nhận vào một tham số là:

```
private time: TimeService
```

Khai báo như thế thì Angular sẽ tạo một đối tượng thuộc lớp TimeService cho lớp SampleComponent với tên là time, và kể từ bây giờ chúng ta có thể dùng đối tượng time giống như một thuộc tính bình thường của một lớp luôn. Do đó ngay trong phương thức khởi tạo chúng ta gọi phương thức getTime () để lấy thời gian

### **9.3 Viết service dưới dạng dùng chung**

Nếu chúng ta dùng service ở nhiều nơi cùng lúc và không muốn khai báo nhiều lần, component nào cũng phải tiêm nó vào. Thì lúc này có thể khai báo service ở phần providers của module hoặc component luôn.

```
@NgModule({
 declarations: [
 AppComponent,
 SampleComponent
],
 imports: [
```

```

 BrowserModule
],
 providers: [TimeService],
 bootstrap: [AppComponent]
})
export class AppModule { }

```

**Chú ý:** Kể từ version của Angular 6, các service không cần đăng kí ở trong module mà chúng ta có thể sử dụng từ khóa `providedIn: 'root'` để xác định tầm ảnh hưởng của service, khi sử dụng cú pháp này mặc định service có thể sử dụng bất cứ đâu trong app, nó tương ứng với việc service đó được import ngay ở AppModule

Như vậy nếu ta khai báo service như thế này

```

import { Injectable } from '@angular/core';
@Injectable({
 providedIn: 'root',
})
export class TimeService {
 constructor() { }
 getTime(){
 return `${new Date().getHours()} : ${new Date().getMinutes()} : ${new
 Date().getSeconds()}`;
 }
}

```

thì service có thể sử dụng bất kỳ đâu trong app mà không cần phải chỉ ra trong providers.

**Chú ý:** Đọc thêm Dependency Injection trong tài liệu:

<https://www.tiepphan.com/thu-nghiem-voi-angular-dependency-injection-trong-angular/>

## 9.4 HttpClient

Hầu hết các ứng dụng giao tiếp với backend thông qua giao thức HTTP. Những trình duyệt hiện đại bây giờ đều hỗ trợ 2 API để tạo ra requests HTTP: XMLHttpRequest và "fetch()" API. Với HTTPCLIENT, @angular/common/http cung cấp một API đơn giản HTTP cho ứng dụng angular, xây dựng một interface

XMLHttpRequest để giao tiếp với trình duyệt. Sự bổ sung của HttpClient là nó hỗ trợ khả năng kiểm tra tốt hơn, các request và response xử lý tốt hơn, hỗ trợ tốt hơn và cả việc handle error nữa

#### **9.4.1 Setup: installing the module**

Để dùng HttpClient, bạn cần install HttpClientModule và provides nó vào app module..

app.module.ts:

```
import {NgModule} from '@angular/core';
import {BrowserModule} from '@angular/platform-browser';
// Import HttpClientModule from @angular/common/http
import {HttpClientModule} from '@angular/common/http';
@NgModule({
 imports: [
 BrowserModule,
 // Include it under 'imports' in your application module
 // after BrowserModule.
 HttpClientModule,
],
})
export class AppModule {}
```

Chỉ cần import HttpClientModule là bạn có thể sử dụng bằng cách inject HttpClient vào component hay services.

#### **9.4.2 Making a request for JSON data**

Hầu hết các request đến từ backend hiện nay đều là request JSON data. Ví dụ: bạn có một API phụ trách một danh sách các item, /api/items sẽ trả về một object JSON như sau:

```
{
 "results": [
 "Item 1",
 "Item 2",
]
}
```

Phương thức `get()` của `HttpClient` sẽ truy cập dữ liệu này một cách đơn giản.

```
@Component(...)
export class MyComponent implements OnInit {
 results: string[];

 // Inject HttpClient into your component or service.
 constructor(private http: HttpClient) {}

 ngOnInit(): void {
 // Make the HTTP request:
 this.http.get<any>('/api/items').subscribe(res => {
 // Read the result field from the JSON response.
 this.results = res['results'];
 // this.results = res.results;
 });
 }
}
```

#### 9.4.3 Error handling

Điều gì xảy ra nếu request server fails hoặc do không có mạng kết nối đến server. `HttpClient` sẽ trả về lỗi thay cho một success response. Để handle nó bạn hãy thêm một error handler vào `.subscribe()`;

```
http
 .get<any>('/api/items')
 .subscribe(
 // Successful responses call the first callback.
 res => {...},
 // Errors will call this callback instead:
 err => {
 console.log('Something went wrong!');
 }
);
```

#### 9.4.4 Getting error details

Phát hiện ra lỗi là một chuyện nhưng chưa đủ, tốt hơn nếu biết lỗi thực sự xảy ra là gì. Thống số err trong callback là một `HttpErrorResponse` có chứa những thông tin về lỗi này.

Có 2 loại lỗi có thể xảy ra. Nếu backend trả về unsuccessful (404, 500,...) thì `httpClient` sẽ nhận về một error. Nhưng nếu lỗi xảy ra trên client hoặc do mất kết nối trường truyền thì một Error cũng sẽ được thrown ra nhưng không là một thể hiện của `HttpErrorResponse`

Trong cả 2 trường hợp bạn có thể xác định lỗi thông qua `HttpErrorResponse`.

```
http
.get<any>('/api/items')
.subscribe(
 data => {...},
 (err: HttpErrorResponse) => {
 if (err.error instanceof Error) {
 // A client-side or network error occurred. Handle it accordingly.
 console.log('An error occurred:', err.error.message);
 } else {
 // The backend returned an unsuccessful response code.
 // The response body may contain clues as to what went wrong,
 console.log(`Backend returned code ${err.status}, body was: ${err.error}`);
 }
 }
);
```

Một cách để giải quyết lỗi là gửi lại request:. Có một toán tử `.retry()`, nó sẽ tự động subscribe lại vào `Observable`. Đầu tiên cần import nó:

```
import 'rxjs/add/operator/retry';
```

Sau đó bạn dùng nó với HTTP Observables :

```
http
.get<ItemsResponse>('/api/items')
 // Retry this request up to 3 times.
```



```
.retry(3)
// Any errors after the 3rd retry will fall through to the app.
.subscribe(...);
```

#### 9.4.5 Requesting non-JSON data

Không phải tất cả các API đều trả về JSON. Giả sử bạn muốn trả về một file text, httpclient sẽ expect một response textual.

```
http
.get('/textfile.txt', {responseType: 'text'})
// The Observable returned by get() is of type Observable<string>
// because a text response was specified. There's no need to pass
// a <string> type parameter to get().
.subscribe(data => console.log(data));
```

#### 9.4.6 Sending data to the server Making a POST request:

Để gửi dữ liệu lên server sẽ dùng phương thức POST . Cách sử dụng phương thức Post cũng đơn giản như Get.

```
const body = {name: 'Brad'};
http
.post('/api/developers/add', body)
// See below - subscribe() is still necessary when using post().
.subscribe(...);
```

#### 9.4.7 Configuring other parts of the request:

URL và Body Param là những thông số cấu hình thông thường bạn hay dùng để gửi lên server. Ngoài ra cũng có nhiều khía cạnh khác bạn có thể cấu hình để request lên server. Headers Thông thường người ra sẽ thêm Authorization vào header của request. Bạn có thể làm như sau:

```
http
.post('/api/items/add', body, {
 headers: new HttpHeaders().set('Authorization', 'my-auth-token'),
})
.subscribe(...);
```

#### 9.4.8 URL Parameters

Thêm URL param có cách sử dụng cũng giống như trên. Để gửi id param với value là 3 bạn làm như sau:

```
http
.post('/api/items/add', body, {
 params: new HttpParams().set('id', '3'),
})
.subscribe();
```

Theo cách này bạn sẽ có POST request tới URL như sau: /api/items/add?id=3.

#### 9.4.9 Advanced usage

Từ đầu tới giờ chỉ là các chức năng cơ bản trong @angular/common/http nhưng đôi khi bạn cần thiết phải làm nhiều hơn là gửi yêu cầu và lấy dữ liệu.

Intercepting all requests or responses: interceptor sẽ xử lý tất cả các request trước khi trả về cho Observable, nó sẽ là trung gian ở giữa chế biến dữ liệu rồi mới chuyển đi cho Subscriber. Writing an interceptor. Để thực hiện một interceptor, bạn cần khai báo một class implements HttpInterceptor, và một function single intercept().

```
import {Injectable} from '@angular/core';
import {HttpEvent, HttpInterceptor, HttpHandler, HttpRequest} from
 '@angular/common/http';
import {Observable} from 'rxjs/Observable';
@Injectable()
export class NoopInterceptor implements HttpInterceptor {
 intercept(req: HttpRequest<any>, next: HttpHandler):
 Observable<HttpEvent<any>> {
 return next.handle(req);
 }
}
```

**Providing your interceptor:**

```
import {NgModule} from '@angular/core';
import {HTTP_INTERCEPTORS} from '@angular/common/http';
import {NoopInterceptor} from 'noop.interceptor.ts';
@NgModule({
```

```
providers: [{
 provide: HTTP_INTERCEPTORS,
 useClass: NoopInterceptor,
 multi: true,
}],
)

export class AppModule {}
```

### **Immutability**

Interceptors sẽ chặn và kiểm tra dữ liệu của response nhưng sẽ không làm thay đổi nó. Bạn có thể thực hiện điều này bằng cách tạo ra bản sao:

```
intercept(req: HttpRequest<any>, next: HttpHandler):
Observable<HttpEvent<any>> {
 // This is a duplicate. It is exactly the same as the original.
 const dupReq = req.clone();
 // Change the URL and replace 'http://' with 'https://'
 const secureReq = req.clone({url: req.url.replace('http://', 'https://')});
}
```

Bạn có thể thấy clone() được sử dụng để tạo ra một bản copy với dữ liệu được thêm.

### **Configuring custom cookie/header names:**

Nếu backend của bạn sử dụng các token và cookie riêng để xử lý, bạn có thể dùng HttpClientXsrfModule.withOptions() để thay đổi giá trị mặc định.

```
imports: [
 HttpClientModule,
 HttpClientXsrfModule.withOptions({
 cookieName: 'My-Xsrf-Cookie',
 headerName: 'My-Xsrf-Header',
 }),
]
```

## **Bài 10: Thảo luận thiết kế và cài đặt website giới thiệu và đặt hàng online(buổi 2)**

### **Nội dung thảo luận:**

- Thiết kế chi tiết cơ sở dữ liệu trên SQL Server
- Thiết kế service để thao tác dữ liệu với server
  - Thiết kế service post dữ liệu
  - Thiết kế service get dữ liệu
  - ...
- Thiết kế và cài đặt web API trên .NET Core để thao tác dữ liệu cho các chức năng đề ra ở buổi 1

**Bài 11: Thảo luận thiết kế và cài đặt website giới thiệu và đặt hàng online(buổi 3)**

**Nội dung thảo luận:**

- Thiết kế và cài đặt các trang khác
  - Trang giỏ hàng
  - Trang tin tức
  - Trang đăng ký

## Bài 12: Phân quyền người dùng

### 12.1 JWT là gì?

JWT là Json Web Token là một tiêu chuẩn mở (RFC 7519) định nghĩa cách thức truyền tin an toàn giữa các thành viên bằng một đối tượng JSON. Thông tin này có thể được xác thực và đánh dấu tin cậy vào nhờ "chữ ký của nó". Phần chữ ký của JWT sẽ được mã hóa lại bằng HMAC hoặc RSA.

### 12.2 Cách thức phân quyền người dùng trong Angular

#### Tạo Guard

Guard được sử dụng để ngăn người dùng không có quyền truy cập đến các routes bị hạn chế, nó được sử dụng trong app.routing.ts.

```
import { Injectable } from '@angular/core';
import { Router, CanActivate } from '@angular/router';
@Injectable()
export class AuthGuard implements CanActivate {
 constructor(private router: Router) { }
 canActivate() {
 if (localStorage.getItem('currentUser')) {
 // logged in so return true
 return true;
 }
 // not logged in so redirect to login page
 this.router.navigate(['/login']);
 return false;
 }
}
```

#### Tạo model

Tạo model cho người dùng là một lớp nhỏ định nghĩa các thuộc tính của người dùng

```
export class User {
 username: string;
 password: string;
 firstName: string;
```

```

 lastName: string;
 }

```

### Angular authentication service

Authenticaiton service dùng để đăng nhập và đăng xuất ra khỏi ứng dụng, để đăng nhập nó gửi thông tin user đến api và kiểm tra xem nếu có jwt token trong response thì đăng nhập thành công và thông tin chi tiết về user được lưu trong local storage và token được thêm vào http authorization header cho tất cả các request tạo bởi http service. Thuộc tính token được sử dụng bởi các dịch vụ khác trong ứng dụng để thiết lập quyền của các yêu cầu http được thực hiện bảo mật cho.... ( The token property is used by other services in the application to set the authorization header of http requests made to secure api endpoints.) Thông tin của current user đã đăng nhập hiện tại được lưu trong local storage vì vậy user sẽ có trạng thái đã đăng nhập tùy họ refresh trình duyệt hoặc giữa các sessions của trình duyệt trừ khi đăng xuất. Nếu không muốn cho user giữ trạng thái đăng nhập trong khi refresh/session có thể thay đổi bằng cách dùng phương pháp lưu khác ngoài local storage để giữ thông tin của current user ví dụ như session storage hoặc root scope.

```

import { Injectable } from '@angular/core';
import { Http, Headers, Response } from '@angular/http';
import { Observable } from 'rxjs';
import 'rxjs/add/operator/map';
@Injectable()
export class AuthenticationService {
 public token: string;

 constructor(private http: Http) {
 // set token if saved in local storage
 var currentUser = JSON.parse(localStorage.getItem('currentUser'));
 this.token = currentUser && currentUser.token;
 }

 login(username: string, password: string): Observable<boolean> {
 return this.http.post('/api/authenticate', JSON.stringify({ username:
username, password: password })))
 .map((response: Response) => {

```

```
// login successful if there's a jwt token in the response
let token = response.json() && response.json().token;
if (token) {
 // set token property
 this.token = token;
 // store username and jwt token in local storage to keep user logged in
 // between page refreshes
 localStorage.setItem('currentUser', JSON.stringify({ username:
username, token: token }));
 // return true to indicate successful login
 return true;
} else {
 // return false to indicate failed login
 return false;
}
});
}
logout(): void {
 // clear token remove user from local storage to log user out
 this.token = null;
 localStorage.removeItem('currentUser');
}
}
```

### **Angular User Service**

User service chứa một method để lấy tất cả người dùng từ api.

```
import { Injectable } from '@angular/core';
import { Http, Headers, RequestOptions, Response } from '@angular/http';
import { Observable } from 'rxjs';
import 'rxjs/add/operator/map';
import { AuthenticationService } from '../_services/index';
import { User } from '../_models/index';
```



```

@Injectable()
export class UserService {
 constructor(
 private http: Http,
 private authenticationService: AuthenticationService) {
 }
 getUsers(): Observable<User[]> {
 // add authorization header with jwt token
 let headers = new Headers({ 'Authorization': 'Bearer ' +
this.authenticationService.token });
 let options = new RequestOptions({ headers: headers });
 // get users from api
 return this.http.get('/api/users', options)
 .map((response: Response) => response.json());
 }
}

```

### Home component template

View mặc định cho thành phần home.

```

<div class="col-md-6 col-md-offset-3">
 <h1>Home</h1>
 <p>You're logged in with JWT!!</p>
</div>

 Users from secure api end point:

 <li *ngFor="let user of users">{{user.firstName}} {{user.lastName}}

</div>

 <p><a [routerLink]="['/login']">Logout</p>
</div>

```

### Home component

Định nghĩa một thành phần angular để định nghĩa một thành phần lấy tất cả các users từ user service

```
import { Component, OnInit } from '@angular/core';
import { User } from '../_models/index';
import { UserService } from '../_services/index';
@Component({
 moduleId: module.id,
 templateUrl: 'home.component.html'
})
export class HomeComponent implements OnInit {
 users: User[] = [];
 constructor(private userService: UserService) { }
 ngOnInit() {
 // get users from secure api end point
 this.userService.getUsers()
 .subscribe(users => {
 this.users = users;
 });
 }
}
```

### Login component template

Chứa form đăng nhập với username và password. nó hiển thị tin nhắn xác thực với trường không hợp lệ khi submit form. Khi submit form thì hàm login() sẽ được gọi

```
<div class="col-md-6 col-md-offset-3">
 <div class="alert alert-info">
 Username: test

 Password: test
 </div>
 <h2>Login</h2>
 <form name="form" (ngSubmit)="f.form.valid && login()" #f="ngForm"
 novalidate>
```

```

<div class="form-group" [ngClass]="{'has-error': f.submitted &&
!username.valid }">
 <label for="username">Username</label>
 <input type="text" class="form-control" name="username"
[(ngModel)]="model.username" #username="ngModel" required />
 <div *ngIf="f.submitted && !username.valid" class="help-
block">Username is required</div>
</div>

<div class="form-group" [ngClass]="{'has-error': f.submitted &&
!password.valid }">
 <label for="password">Password</label>
 <input type="password" class="form-control" name="password"
[(ngModel)]="model.password" #password="ngModel" required />
 <div *ngIf="f.submitted && !password.valid" class="help-
block">Password is required</div>
</div>

<div class="form-group">
 <button [disabled]="loading" class="btn btn-primary">Login</button>

</div>
<div *ngIf="error" class="alert alert-danger">{{error}}</div>
</form>
</div>

```

## Login component

Login component sẽ sử dụng authentication service để đăng nhập và đăng xuất khỏi ứng dụng. nó tự động ghi lại người dùng khi nó khởi tạo (ngOnInit) để trang đăng nhập cũng có thể sử dụng để đăng xuất

```

import { Component, OnInit } from '@angular/core';
import { Router } from '@angular/router';
import { AuthenticationService } from '../_services/index';

@Component({
 moduleId: module.id,
 templateUrl: 'login.component.html'
})
export class LoginComponent implements OnInit {
 model: any = {};
 loading = false;
 error = '';

 constructor(
 private router: Router,
 private authenticationService: AuthenticationService) {}

 ngOnInit() {
 // reset login status
 this.authenticationService.logout();
 }

 login() {
 this.loading = true;
 this.authenticationService.login(this.model.username, this.model.password)

```

```
.subscribe(result => {
 if (result === true) {
 // login successful
 this.router.navigate(['\^']);
 } else {
 // login failed
 this.error = 'Username or password is incorrect\';
 this.loading = false;
 }
});
}
}
```

### App component template

Đây là root component template của ứng dụng, nó chứa một router-outlet điều hướng cho hiển thị về nội dung của mỗi thành phần dựa trên đường dẫn hiện tại.

```
<div class="jumbotron">
 <div class="container">
 <div class="col-sm-8 col-sm-offset-2">
 <router-outlet></router-outlet>
 </div>
 </div>
</div>
```

### App component

Đây là root component của ứng dụng, nó định nghĩa thẻ gốc của ứng dụng là `<app></app>` với thuộc tính selector.

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

@Component({
 moduleId: module.id,
 selector: 'app',
 templateUrl: 'app.component.html'
})
```

---

```
export class AppComponent { }
```

### App module

App module định nghĩa một root module của ứng dụng cùng với siêu dữ liệu về module. Đây là nơi để thêm fake backend provider cho ứng dụng. Để chuyển sang một backend thực sự, chỉ cần xóa các nhà cung cấp nằm trong chú thích "// providers used to create fake backend".

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { FormsModule } from '@angular/forms';
import { HttpClientModule } from '@angular/http';
// used to create fake backend
import { fakeBackendProvider } from './_helpers/index';
import { MockBackend, MockConnection } from '@angular/http/testing';
import { BaseRequestOptions } from '@angular/http';
import { AppComponent } from './app.component';
import { routing } from './app.routing';
import { AuthGuard } from './_guards/index';
import { AuthenticationService, UserService } from './_services/index';
import { LoginComponent } from './login/index';
import { HomeComponent } from './home/index';
@NgModule({
 imports: [
 BrowserModule,
 FormsModule,
 HttpClientModule,
 routing
],
 declarations: [
 AppComponent,
 LoginComponent,
 HomeComponent
],
```

```
providers: [
 AuthGuard,
 AuthenticationService,
 UserService,
 // providers used to create fake backend
 fakeBackendProvider,
 MockBackend,
 BaseRequestOptions
],
bootstrap: [AppComponent]
})

export class AppModule { }
```

### App routing

App routing định nghĩa đường định tuyến của ứng dụng, mỗi route chứa một đường dẫn và liên kết với các component khác.

```
import { Routes, RouterModule } from '@angular/router';
import { LoginComponent } from './login/index';
import { HomeComponent } from './home/index';
import { AuthGuard } from './_guards/index';

const appRoutes: Routes = [
 { path: 'login', component: LoginComponent },
 { path: '', component: HomeComponent, canActivate: [AuthGuard] },
 // otherwise redirect to home
 { path: '**', redirectTo: '' }
];

export const routing = RouterModule.forRoot(appRoutes);
```

### Main file

Là điểm vào của ứng dụng, nơi mà app module được khai báo với các dependencies và chứa các cấu hình và khởi tạo logic khi ứng dụng load lần đầu tiên.

```
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
import { AppModule } from './app.module';

platformBrowserDynamic().bootstrapModule(AppModule);
```

**Bài 13: Thảo luận thiết kế và cài đặt website giới thiệu và đặt hàng online(buổi 4)**

**Nội dung thảo luận:**

- Thiết kế các chức năng cho phần quản trị
- Thiết kế và cài đặt một số chức năng
  - o Xác thực người dùng
  - o Quản trị người dùng

**Bài 14: Thảo luận thiết kế và cài đặt website giới thiệu và đặt hàng online(buổi 5)**

**Nội dung thảo luận:**

- Thiết kế và cài đặt một số chức năng khác
  - o Quản lý danh mục
  - o Quản lý sản phẩm
  - o Quản lý đơn hàng

**Bài 15: Tổng kết bài học**

**Nội dung thảo luận:**

- Tổng kết các kiến thức cốt lõi của môn học
- Tổng kết lại một số kinh nghiệm khi triển khai một dự án với Angular + Net Core & SQL Server



## **TÀI LIỆU THAM KHẢO**

1. <https://www.tutorialspoint.com/typescript/index.htm>
2. <https://www.tutorialsteacher.com/typescript/>
3. <https://www.typescriptlang.org/index.html>
4. <https://training-course-material.com/training/Typescript>
5. <https://angular.io/>
6. <https://jasonwatmore.com/>
7. <https://www.typescripttutorial.net/>
8. <https://dotnettutorials.net/course/angular-tutorials/>

Một số tài liệu khác đính kèm trong thư mục bài giảng