

MỤC LỤC

1. Giới thiệu Angular 4.....	3
1.1 Giới thiệu.....	3
1.2 Phân tích một ứng dụng Angular 4	6
1.3 Ứng dụng mẫu Angular 4.....	7
2. Giới thiệu về Components	8
2.1 Giới thiệu.....	8
2.2 Component là gì ?	8
2.3 Tạo một Component Class	10
2.4 Định nghĩa Metadata với một Decorator	11
2.5 Import.....	12
2.6 Bootstrapping	13
2.7 Tổng kết.....	15
3. Tempates, Interpolation, and Directives.....	18
3.1 Giới thiệu.....	18
3.2 Xây dựng một mẫu (Template).....	19
3.3 Xây dựng Component và sử dụng như một Directive.....	20
3.4 Ràng buộc dữ liệu với Interpolation	21
3.5 Thêm logic với directive ngIf	22
3.6 Tổng kết.....	25
4. Data binding & Pipes	27
4.1 Giới thiệu.....	27
4.2 Property Binding	28
4.3 Xử lý sự kiện với Event Binding	28
4.4 Xử lý Input với Two-way binding	29
4.5 Biến đổi dữ liệu với Pipes	31
4.6 Tổng kết.....	32
5. Component chuyên sâu	34
5.1 Giới thiệu.....	34
5.2 định nghĩa một interface.....	35
5.3 Đóng gói Component Styles	37
5.4 Sử dụng Lifecycle Hooks	37
5.5 Xây dựng những Custom Pipe	40
5.6 Tổng kết.....	42
6. Xây dựng những component lồng nhau	45
6.1 Giới thiệu.....	45
6.2 Xây dựng một component lồng nhau (nested component).....	46
6.3 Sử dụng một Nested Component.....	49

6.4	Truyền dữ liệu tới một Nested Component (@Input).....	51
6.5	Truyền dữ liệu từ một Component (@Output)	52
6.6	Tổng kết.....	53
7.	Service và Dependency Injection	55
7.1	Giới thiệu.....	55
7.2	Nó làm việc như thế nào?	55
7.3	Xây dựng một service	56
7.4	Đăng ký service	58
7.5	Injecting the Service.....	58
7.6	Tổng kết.....	61
8.	Truy xuất dữ liệu bằng HTTP	63
8.1	Giới thiệu.....	63
8.2	Observables và Reactive Extensions.....	63
8.3	Gửi một yêu cầu HTTP.....	64
8.4	Xử lý ngoại lệ.....	66
8.5	Đăng ký một Observable.....	67
8.6	Tổng kết.....	68
9.	Khái niệm cơ bản về Định hướng (Navigation) và Định tuyến (Routing)	69
9.1	Giới thiệu.....	69
9.2	Cách Routing hoạt động	70
9.3	Cấu hình routes.....	71
9.4	Kết nối các Route tới các Action.	73
9.5	Xếp chỗi các View	73
9.6	Tổng kết.....	74

1. GIỚI THIỆU ANGULAR 4

1.1 GIỚI THIỆU

Cho dù bạn là người mới đến với Angular hoặc đã biết qua Angular và mới tiếp cận với Angular 4. Chào mừng bạn đến với khoá học Angular 4 từ **Janeto Traning Center**

Cuốn sách này cung cấp những điều cơ bản mà bạn cần để bắt đầu xây dựng một ứng dụng Angular 4. Khi chúng ta hoàn thành khoá học này chúng ta sẽ khám phá ra nhiều tính năng của Angular và có câu trả lời cho những câu hỏi quan trọng. Ví dụ như

- Một “component” là gì?
- Chúng ta sẽ viết mã HTML ở đâu trong source?
- Khi nào chúng ta nên sử dụng “data binding”?
- Tại sao chúng ta lại cần một “service”?
- Làm thế nào để chúng ta xây dựng được một ứng dụng Angular?

Xuyên suốt cuốn sách sẽ là những hướng dẫn cho bạn đi đúng hướng, giúp cho cuộc hành trình các bạn tiếp cận với Angular sẽ đơn giản và hiệu quả hơn.

Để bắt đầu hành trình này, bạn sẽ cần có một cái nhìn toàn cảnh về Angular. Đơn giản mà nói, Angular là một Javascript framework để xây dựng những ứng dụng phía client sử dụng HTML, CSS và một ngôn ngữ lập trình như Javascript.

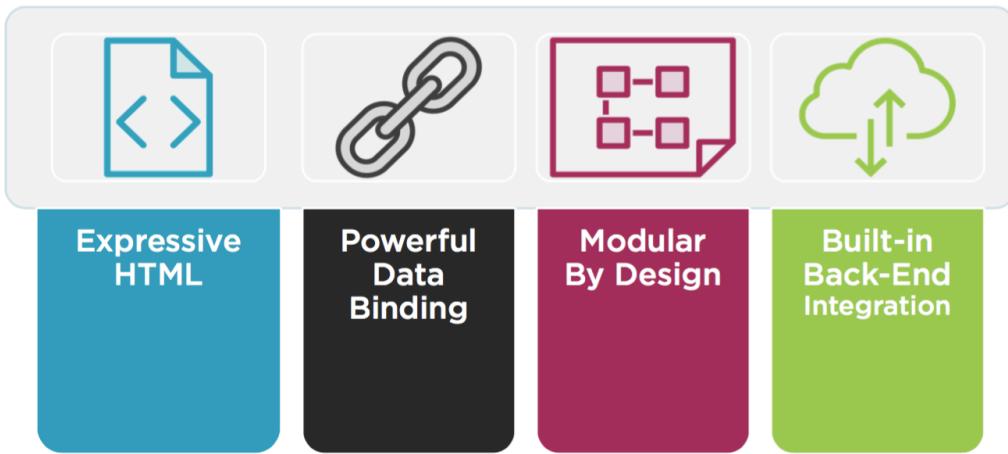
Angular Is ...



**A JavaScript framework
For building client-side applications
Using HTML, CSS and JavaScript**

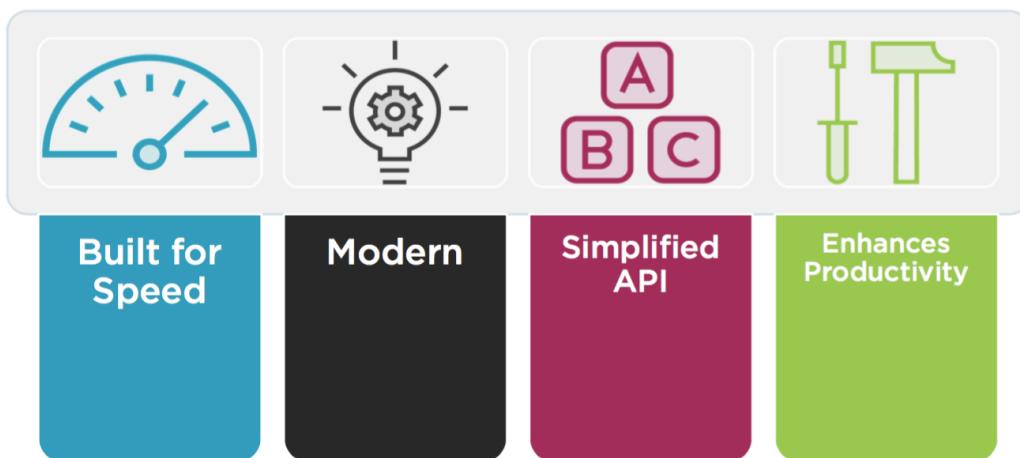
Tại sao lại là Angular mà không phải một Javascript framework khác? Angular làm cho HTML của chúng ta trở nên linh hoạt hơn, Nó làm cho code HTML của chúng ta trở nên mạnh mẽ hơn với những đặc trưng như điều kiện “if” , vòng lặp “for” và những biến địa phương “local variables”. Angular có cơ chế binding data mạnh mẽ, chúng ta có thể dễ dàng hiển thị các trường từ data model của chúng ta, theo dõi những thay đổi và cập nhật lại từ người dùng. Angular hoạt động theo thiết kế module (mô đun). Ứng dụng của chúng ta sẽ được xây dựng từ những khôi module độc lập, làm cho việc xây dựng sẽ dễ dàng và có thể tái sử dụng được nội dung. Hơn thế nữa, Angular hỗ trợ việc giao tiếp với những back-end service. Điều này sẽ dễ dàng cho việc tích hợp những back-end service để việc giải quyết các bài toán logic nhanh chóng. Cuối cùng, Angular là cực kỳ phổ biến với hàng triệu nhà phát triển đang sử dụng nó.

Why Angular?



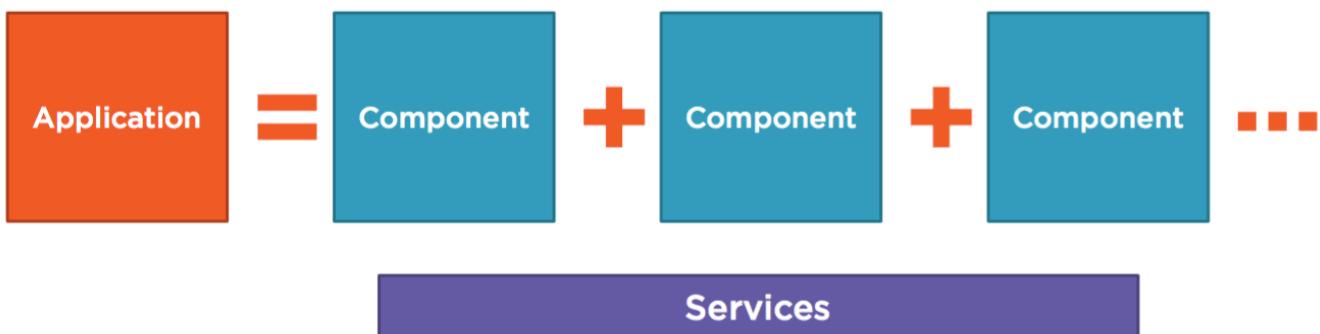
Nếu bạn đang là một nhà phát triển web dựa trên Angular 1. Tại sao bạn cần đến Angular 4 nữa? Angular 4 nhanh hơn Angular 1 và liên tục được nâng cao hiệu năng. Angular 4 hiện đại, nó sử dụng những tính năng của Javascript mới nhất (es6, es7) và hơn thế nữa các Class, Module, Decorators hỗ trợ nhiều trình duyệt hiện tại Edge, Chrome, Firefox và cả IE thần thánh. Angular 4 đơn giản hóa các API, nó có ít directive hơn nên dễ nhớ dễ học hơn, binding data đơn giản, khái niệm tổng thể cũng đơn giản hơn. Phát triển ứng dụng bằng Angular 4 bạn cũng sẽ nhận ra những cải tiến về năng xuất, điều này cũng nhờ sự nhất quán của mô hình xây dựng các khối để hình thành ứng dụng.

Why Angular 4?

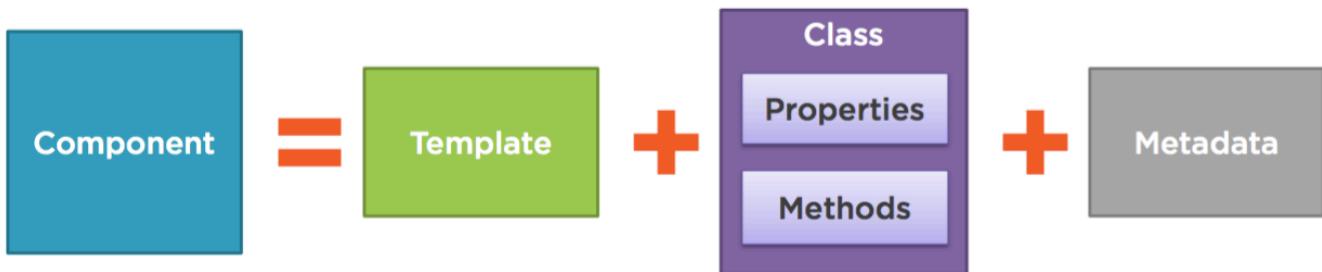


1.2 PHÂN TÍCH MỘT ỨNG DỤNG ANGULAR 4

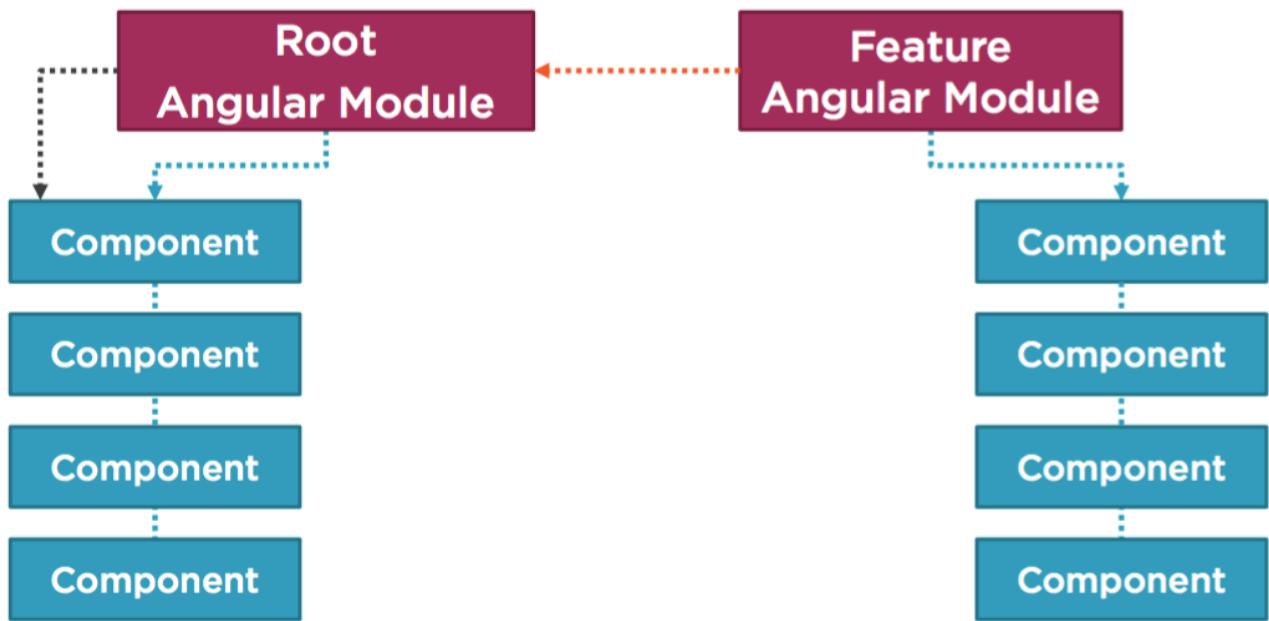
Trong Angular 4, một ứng dụng là một tập hợp những component (thành phần) và một vài service (dịch vụ) cung cấp chức năng trên những component đó. Vậy, câu hỏi tiếp theo rõ ràng là một component của Angular 4 là gì ?



Mỗi một component bao gồm một mẫu HTML cái mà sẽ render ra một “frame” giao diện người dùng. Thêm vào đó là một Class để code những gì liên kết với view. Class chứa những thuộc tính, những phần tử dữ liệu có sẵn để phục vụ cho các view và các phương thức thực hiện những hành động cho view, chẳng hạn như sự phản hồi của một nút bấm. Một component cũng có metadata cung cấp thêm những thông tin của component cho Angular. Metadata này để xác định Class là một Angular component. Vậy tóm lại, một component sẽ hình thành từ một view xác định từ một mẫu HTML, Code xử lý sẽ được định nghĩa với một Class và thông tin bổ xung cho Class thì được xác định bằng metadata. Chúng ta sẽ tìm hiểu chi tiết hơn trong những bài sắp tới.



Chúng ta đã biết một component là gì, làm thế nào để ghép chúng lại với nhau để trở thành một ứng dụng? Chúng ta sẽ có một định nghĩa Angular modules (mô-dun). Angular modules giúp chúng ta tổ chức ứng dụng Angular của chúng ta thành một khối gắn kết các chức năng. Mỗi ứng dụng Angular có ít nhất một module, module này gọi là **Root Angular Module** (mô-dun gốc). Một ứng dụng thực tế thì có thể có nhiều hơn một module. Chúng ta sẽ hiểu rõ hơn về Angular module đọc theo hành trình của chúng ta.



1.3 ỨNG DỤNG MẪU ANGULAR 4.

Để hiểu thêm về những chức năng trong Angular 4, chúng ta sẽ xây dựng một ứng dụng đơn giản theo từng bước dọc theo nội dung quyển sách này.

Trước tiên chúng ta hãy xem ứng dụng hoàn thành của chúng ta sẽ hoạt động như thế nào? [Link here](#)

2. GIỚI THIỆU VỀ COMPONENTS

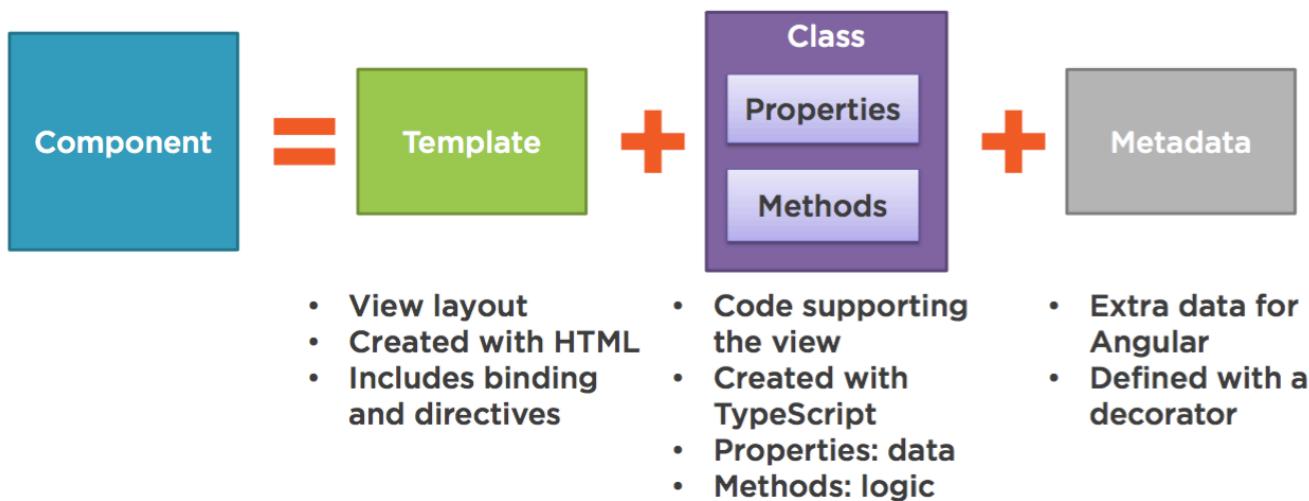
2.1 GIỚI THIỆU

Trong chương trước, chúng ta đã chuẩn bị sẵn sàng môi trường và công cụ cần thiết để bắt đầu xây dựng ứng dụng Angular. Ở chương này chúng ta sẽ đi sâu vào việc xây dựng một component rất cơ bản và tập trung xác định rõ các bộ phận của component, ý nghĩa và mục đích của chúng. Chúng ta có thể nghĩ về ứng dụng Angular của chúng ta là tập hợp các component, chúng ta tạo ra từng component sau đó sắp xếp chúng lại để tạo ra ứng dụng. Nếu mọi việc suôn sẻ, các component làm việc với nhau hài hoà để cung cấp cho người dùng một trải nghiệm hoàn hảo.

Tóm tắt chương này:

- Component là gì ?
- Tạo một Component Class
- Định nghĩa Metadata với Decorator
- Importing những gì chúng ta cần
- Bootstrapping

2.2 COMPONENT LÀ GÌ ?

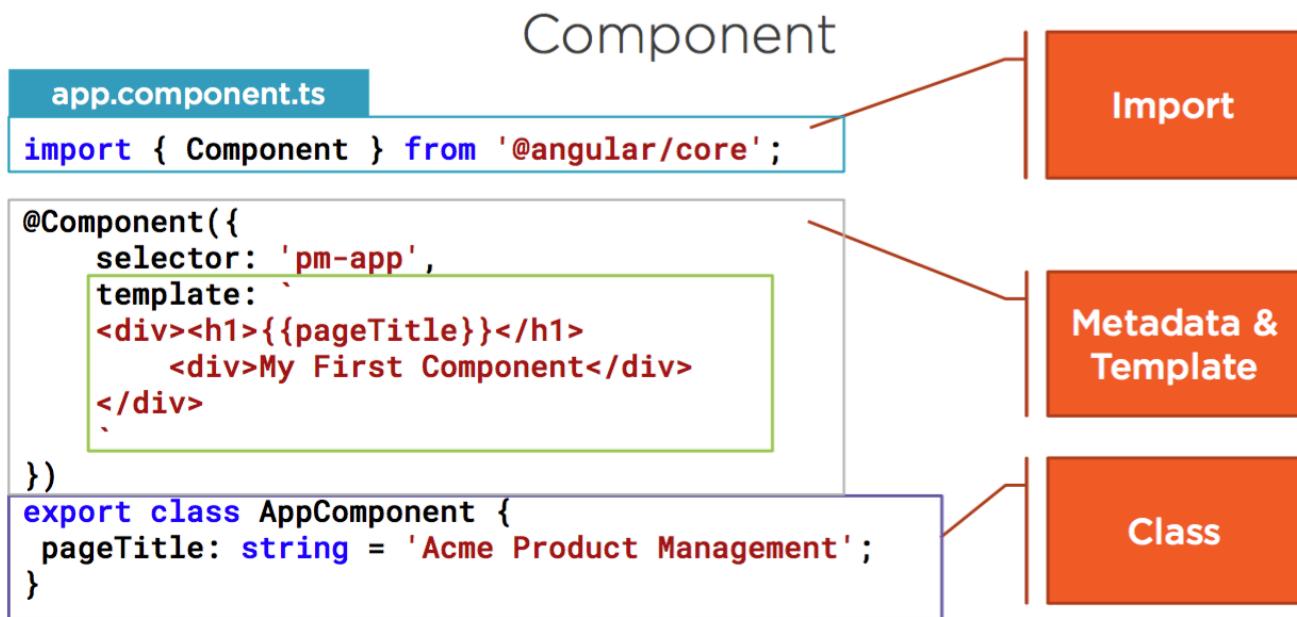


Một Angular Component bao gồm một mẫu (template) cái mà sẽ tạo ra giao diện (layout view) cho người dùng, template này được tạo ra từ HTML. Trên trang HTML này, chúng ta sẽ sử dụng Angular Binding và Directives để tăng sức mạnh cho View. Tôi sẽ giới thiệu về Binding và Directive trong những chương sau.

Thêm vào đó, Component sẽ có một Class để chúng ta liên kết với View. Các Class này được tạo ra bằng TypeScript. Class có chứa các thuộc tính và phương thức, cũng tương tự như định nghĩa Class của lập trình hướng đối tượng.

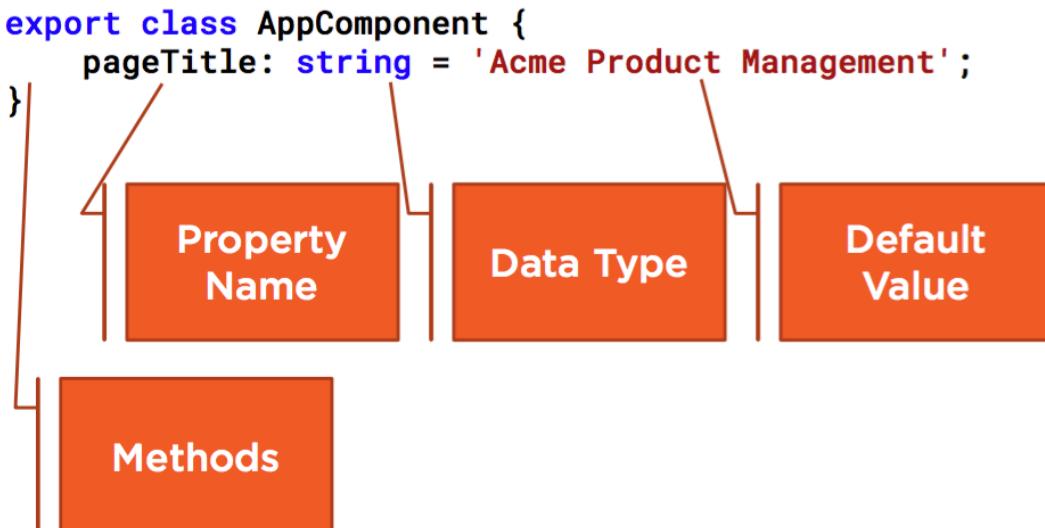
Một phần không thể thiếu khác của Component là Metadata, nó cung cấp thông tin bổ xung cho Angular Component và giúp định nghĩa Class này như một Angular Component. Metadata được định nghĩa bởi một Decorator. Một decorator có thể hiểu đơn giản là phương thức thêm Metadata vào Class.

Dưới đây là một component đơn giản viết bằng Typescript.



2.3 TẠO MỘT COMPONENT CLASS

app.component.ts



Nếu như các bạn đã biết qua về lập trình hướng đối tượng trong các ngôn ngữ như C#, Java, C++ thì việc tạo một class sẽ trông rất quen thuộc. Một class là một cấu trúc cho phép chúng ta tạo những thuộc tính(properties) được xác định bởi những kiểu dữ liệu (data type) và những phương thức (method) cung cấp những chức năng. Chúng ta định nghĩa một class sử dụng một từ khoá "class" và theo đó là tên của class. Thông thường Angular quy ước đặt tên mỗi component với một tên đặc trưng. Sau đó nối các thành phần như hậu tố "Component". Ngoài ra, theo quy ước, Component gốc của ứng dụng được gọi là `AppComponent` như ví dụ trên. Tên class này được sử dụng như tên component khi các component được tham chiếu trong mã. Từ khoá "export" trước class làm cho component đó có thể được sử dụng bởi các component khác của ứng dụng. Và như chúng ta đã được học từ chương trước, file chứa mã bây giờ đã là một ES module. Nó sẽ được tải bởi module loader khác mà không cần thêm một script tag nữa. Bên trong phần thân của class có những thuộc tính và phương thức. Trong ví dụ này chúng ta chỉ có một thuộc tính và không có phương thức. Thuộc tính định nghĩa một phần tử dữ liệu của một class. Chúng ta bắt đầu với tên thuộc tính, theo quy ước là một danh từ mô tả phần tử dữ liệu và viết theo kiểu camelCase, theo đó chữ cái đầu tiên là chữ thường. Trong ví dụ này, nó là tiêu đề của một trang. Sử dụng Typescript's strong type, theo sau tên thuộc tính chúng ta sẽ thêm kiểu dữ liệu của nó. Chúng ta cũng có thể tùy chọn gán cho thuộc tính một giá trị ban đầu như trên ví dụ. Các phương thức thông thường được đặt trong thân của class và sau những thuộc tính. Tên phương thức thường là động từ mô tả hành động mà phương thức đó thực hiện. Những phương thức cũng được viết theo kiểu camelCase. Vậy đó là class, Nhưng một class không đủ để xác định một component. Chúng ta cần định nghĩa một template liên kết với component class này. Làm thế nào để chúng ta cung cấp những thông tin này cho Angular? Chúng ta sẽ theo dõi tiếp theo đây.

2.4 ĐỊNH NGHĨA METADATA VỚI MỘT DECORATOR

app.component.ts

```
@Component({
  selector: 'pm-app',
  template: `
    <div><h1>{{pageTitle}}</h1>
      <div>My First Component</div>
    </div>
  `

})
export class AppComponent {
  pageTitle: string = 'Acme Product Management';
}
```

Một class sẽ trở thành một component khi mà chúng ta cung cấp cho nó metadata. Angular cần metadata đó để hiểu làm thế nào để tổ chức component, xây dựng view và tương tác với component. Chúng ta định nghĩa một metadata của component với Angular component function. Trong typescript chúng ta gắn chức năng đó vào class như một decorator. Một decorator là một hàm cái mà thêm metadata vào class các thành viên của mình, hoặc các đối số của nó. Một decorator là một đặc trưng của ngôn ngữ JavaScript được thực hiện trong Typescript và đề xuất trong ES2016. Phạm vi của decorator là giới hạn trong tính năng mà nó "trang trí". Một decorator luôn được bắt đầu bằng một ký tự @. Angular đã cung cấp những decorator có sẵn. Chúng ta sử dụng để cung cấp thông tin cho Angular. Chúng ta cũng có thể tự xây dựng những decorator của mình, nhưng điều đó thì nằm ngoài phạm vi của chương này. Chúng ta áp dụng một decorator bằng cách để nó nằm ngay trước đặc trưng mà chúng ta muốn. Khi "trang trí" một class như trong ví dụ này, Chúng ta định nghĩa một decorator ngay trên class. Lưu ý không có dấu chấm phẩy ";". Cú pháp này gần giống như các attributes được sử dụng trong các ngôn ngữ khác. Chúng ta sử dụng một decorator để chỉ định một class là một component. Vì decorator là một hàm, nên chúng ta luôn thêm dấu ngoặc đơn cho nó. Ta truyền một đối tượng vào @Component nhưng trên ví dụ. Đối tượng này sẽ chứa nhiều thuộc tính hiện tại chúng ta chỉ sử dụng 2 trong số đó. Nếu chúng ta dự định tham chiếu đến component trong một HTML nào đó, chúng ta chỉ định cho component một selector. Một selector sẽ định nghĩa một directive name. Một directive đơn giản là một custom HTML tag. Bất cứ khi nào directive được sử dụng ở trong HTML, Angular sẽ render component template này. Chúng ta sẽ xem cách mà chúng hoạt động ở trong ví dụ tới. Một component luôn luôn có một template. Ở đây chúng ta sẽ xác định cách bố trí giao diện người dùng cũng như những thành phần sẽ được quản lý bởi component này. Đầu {{ }} trong ví dụ dùng để binding dữ liệu, pageTitle trong thẻ H1 sẽ hiện giá trị thuộc tính pageTitle của class khi mà HTML được render cụ thể sẽ hiện thị "Acme Product Management". Những ví dụ sắp tới sẽ đề cập nhiều hơn về binding, chúng ta sẽ đi đến một phần quan trọng hơn trước khi có thể xây dựng một component cơ bản. Đó là Import.

2.5 IMPORT

Trước khi chúng ta có thể sử dụng một external function (chức năng bên ngoài) hay một class. Chúng ta cần báo cho module loader nơi mà chúng ta tìm nó. Chúng ta làm điều này với một câu lệnh import. Câu lệnh import là một phần của ES2015 và được kế thừa trong TypeScript. Nó là khái niệm tương tự như câu lệnh import trong Java hay C# sử dụng. Câu lệnh import cho phép chúng ta sử dụng những thành phần exported (xuất khẩu) từ module bên ngoài. Hãy nhớ lại cách chúng ta đã viết câu lệnh export class? Điều đó có nghĩa rằng những module khác trong ứng dụng của chúng ta có thể import exported class nếu cần. Chúng ta sẽ sử dụng câu lệnh import xuyên suốt quá trình code để import bất kỳ thư viện bên thứ ba nào, hoặc những module của chúng ta đã viết, hay import từ chính Angular.

Angular Is Modular



Chúng ta có thể import từ Angular, vì Angular cũng chính là một tập hợp những thư viện module. Mỗi thư viện là một module gồm nhiều tính năng liên quan. Khi chúng ta cần thứ gì đó từ Angular, Chúng ta sẽ import một Angular module library (thư viện module). Giống như chúng ta import bất kỳ module bên ngoài khác.

```
app.component.ts
import { Component } from '@angular/core';
@Component({
  selector: 'pm-app',
  template: `
    <div><h1>{{pageTitle}}</h1>
      <div>My First Component</div>
    </div>
  `)
export class AppComponent {
  pageTitle: string = 'Acme Product Management';
}
```

import keyword

Angular library module name

Member name

Trong đoạn mã trên, chúng ta sử dụng **@Component** chức năng từ Angular để định nghĩa class của chúng ta là một component. Chúng ta cần báo cho module loader nơi để tìm function này, vì vậy chúng ta cần một câu lệnh import, một import component từ Angular Core chẳng hạn.

2.6 BOOTSTRAPPING

Chúng ta cần nói với Angular để nạp Component gốc của chúng ta thông qua một quá trình gọi là bootstrapping. Và chúng ta cần phải thiết lập các tập tin index.html để host ứng dụng của chúng ta. Hãy xem tất cả các bước sau:

Single Page Application (SPA)



index.html contains the main page for the application

This is often the only Web page of the application

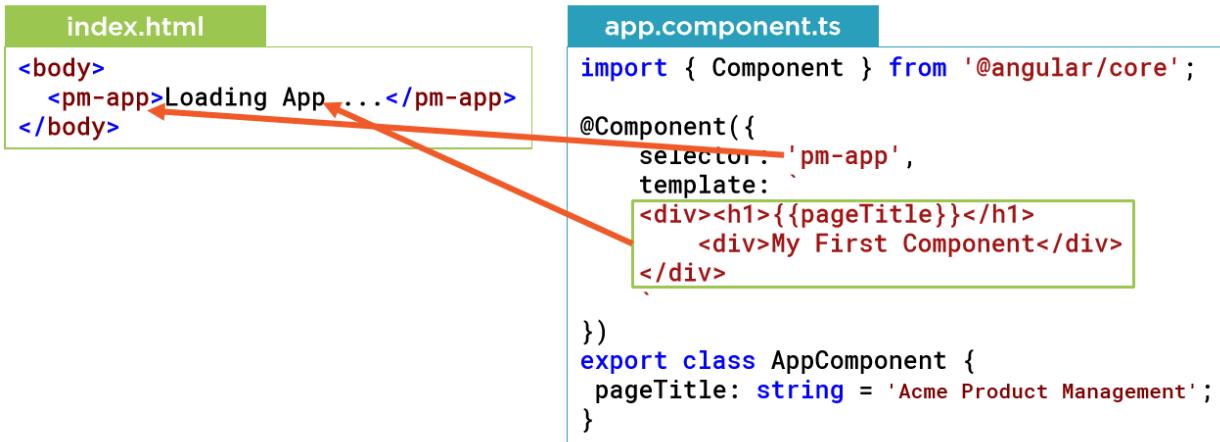
Hence an Angular application is often called a Single Page Application (SPA)

Hầu hết các ứng dụng Angular có một file index.html có chứa trang chính của ứng dụng. File index.html thường chỉ là một trang. Do đó một ứng dụng Angular thường được gọi là một Single Page Application (SPA). Nhưng đừng lo lắng, mặc dù tên gọi là “Ứng dụng một trang” nhưng chúng ta sẽ có rất nhiều trang giống như trong bản demo ở đầu khóa học này mà tôi đã giới thiệu.

Những gì chúng ta làm là thêm những phần của HTML vào trong trang index.html. Hãy xem cách mà chúng làm việc ở đoạn ví dụ dưới đây.

Đoạn code phía bên phải là app.component và selector: ‘pm-app’ sẽ là một chỉ thị trong HTML. Đoạn HTML bên trong template là những gì chúng ta muốn hiển thị.

Phía bên trái là file index.html, nơi mà chúng ta muốn hiển thị app.component. Để chèn những HTML của app component chúng ta chỉ đơn giản thêm selector <pm-app></pm-app> vào nơi mà chúng ta muốn hiển thị chúng. Chúng ta gọi chúng là một directive (chỉ thị). Một directive về cơ bản là một custom element.



Nhưng làm sao để module loader tìm được component để khởi động ứng dụng? Chúng ta sẽ cần định nghĩa một root module, cái mà sẽ khởi động đầu tiên để bắt đầu ứng dụng. Bạn có thể đặt tên module này bất cứ tên nào bạn muốn, thông thường tên module gốc là AppModule.

Đoạn code sau là những gì tối thiểu phải có để tạo ra một AppModule

```

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

@NgModule({
  imports:      [ BrowserModule ],
  declarations: [ AppComponent ],
  bootstrap:   [ AppComponent ]
})
export class AppModule { }

```

- @NgModule để xác định 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.
 - o Imports: import module BrowserModule, module này cần cho mọi ứng dụng chạy trên trình duyệt.
 - o Declarations: Khai báo những component sẽ được sử dụng trong module này.
 - o Bootstrap: Component gốc được Angular tạo ra và chèn vào trong index.html

Để tìm hiểu sâu hơn về Angular Module chúng ta sẽ theo dõi tiếp ở chặng đường tiếp theo, Tất cả những gì bạn cần biết bây giờ là 3 thuộc tính trên.

Tuy nhiên một ứng dụng Angular có thể có nhiều hơn một module, vậy làm sao để Angular biết được sẽ phải khởi chạy module nào đầu tiên?

Chúng ta có rất nhiều cách để khởi chạy (bootstrap) một ứng dụng, điều phụ thuộc vào cách bạn muốn Angular biên dịch ứng dụng như thế nào và nơi bạn muốn chạy ứng dụng là ở đâu? Ở trong khóa học này chúng ta sẽ lần lượt tiếp cận 2 cách biên dịch ứng dụng là JIT và AOT. Đoạn code dưới đây là một ví dụ cụ thể để khởi chạy một ứng dụng bằng JIT.

src/main.ts

```
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
import { AppModule } from './app/app.module';

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

Một điều may mắn là thiết lập file bootstrap này gần như không thay đổi trong suốt quá trình chúng ta code ứng dụng, nên bạn cũng không cần quá bận tâm về nó.

Cuối cùng, bạn cũng đã biết cách để tạo ra một Root Module, chỉ định module này khởi chạy một component mà bạn mong muốn, chèn một chỉ thị (directive) vào index.html để hiển thị giao diện component. Hiện tại chúng ta chỉ có một module, nhưng khi ứng dụng phát triển lớn hơn, chúng ta sẽ chia ứng dụng thành những module tính năng nhỏ hơn, một trong số những module đó sẽ "lazy loaded" tức là module đó chỉ chạy khi bạn thực sự ghé thăm những tính năng đó.

2.7 TỔNG KẾT

Class



Clear name

- Sử dụng kiểu đặt tên PascalCasing
- Thêm “Component” sau tên

export keyword

Dữ liệu trong các thuộc tính

- Kiểu dữ liệu thích hợp
- Giá trị mặc định thích hợp
- Đặt tên thuộc tính kiểu camelCase

Logic trong các phương thức

- Đặt tên phương thức kiểu camelCase

Metadata



Component decorator

- Tiền tố với @; Hậu tố với ()

Selector: Tên component trong HTML

- Tiền tố rõ ràng

Template: HTML của View

- Cú pháp HTML chính xác

Import



Định nghĩa nơi để tìm những member mà component cần

import keyword

Tên member (Member name)

- Đúng chính tả

Đường dẫn tới module (Module path)

- Đặt trong dấu ""
- Đúng chính tả

3. TEMPATES, INTERPOLATION, AND DIRECTIVES

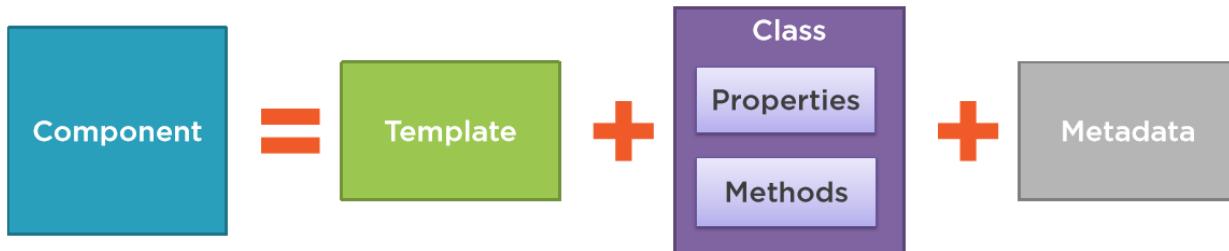
3.1 GIỚI THIỆU

Để xây dựng một giao diện người dùng cho ứng dụng Angular, chúng ta tạo ra một mẫu (template) với HTML, chúng ta sẽ cần Angular data binding và những chỉ thị (directives) để thật sự thấy được sức mạnh mà Angular có thể làm với giao diện người dùng. Trong bài học này chúng ta sẽ tìm hiểu cách tạo ra một giao diện ứng dụng Angular bằng cách sử dụng mẫu (template), những chỉ thị (directives) và ràng buộc dữ liệu (data binding).

Các ứng dụng web tất cả đều có giao diện người dùng. Và Angular làm việc xây dựng giao diện trở nên đơn giản và mạnh mẽ hơn. Angular cho chúng ta cơ chế data binding, vì vậy chúng ta có thể dễ dàng hiển thị thông tin cũng như phản hồi những hành động của người dùng. Với Angular directives chúng ta dễ dàng thêm những logic vào đoạn mã HTML, như câu lệnh IF hay vòng lặp For. Và mới Angular Component chúng ta xây dựng các đoạn giao diện lồng với nhau. Chẳng hạn như ví dụ dưới đây

<http://codepen.io/rajdgreat007/pen/zBgxJG>

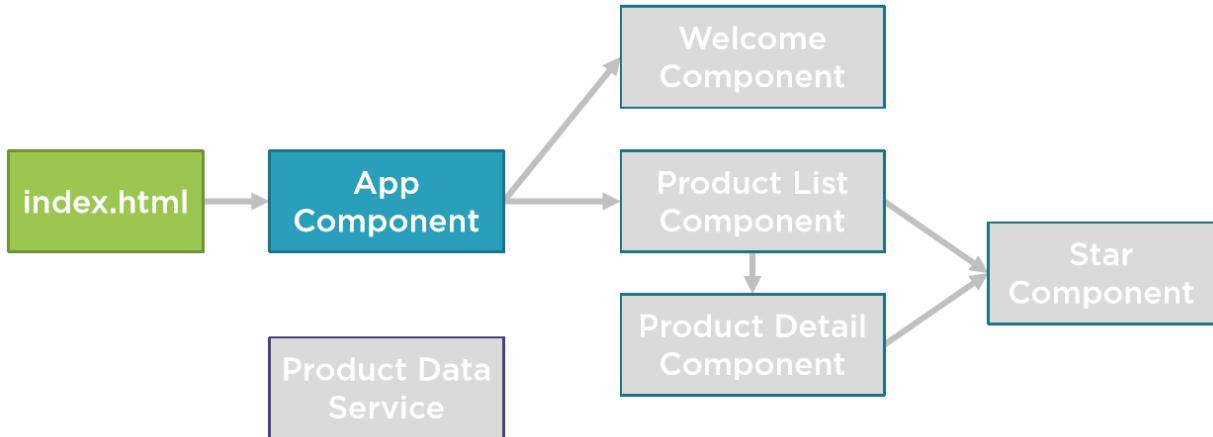
Chúng ta đã thấy rằng Angular component là một khung nhìn (view) được định nghĩa bằng một khuôn mẫu (template). Code liên kết được định nghĩa với một Class và thông tin bổ xung cho class được định nghĩa bởi những metadata thông qua một component decorator. Trong bài học này chúng ta sẽ tập trung vào các kỹ thuật để xây dựng mẫu (template). Chúng ta sẽ tiếp cận những cách khác nhau mà chúng ta có thể xây dựng một mẫu (template) cho component.



Những nội dung chính của bài học này:

- Xây dựng một mẫu (template).
- Sử dụng một component như một chỉ thị (directive).
- Ràng buộc dữ liệu với Interpolation.
- Thêm logic vào chỉ thị (directive).

Hãy nhìn lại kiến trúc ứng dụng mà chúng ta sẽ làm.



Để chúng ta có thể hình dung rõ hơn về những nội dung chính trong bài học, chúng ta sẽ bắt tay vào xây dựng **Product List Component**.

3.2 XÂY DỰNG MỘT MẪU (TEMPLATE).

Trong chương trước, chúng ta đã tạo ra một inline template cho App Component. Chúng ta đã sử dụng thuộc tính “template” để định nghĩa mẫu (template) trực tiếp trong metadata của component. Nhưng đó không phải là cách duy nhất chúng ta có thể tạo ra một template cho những component của chúng ta. Chúng ta có thể sử dụng thuộc tính “template” và định nghĩa một inline template sử dụng một chuỗi trích dẫn đơn giản với dấu nháy đơn hoặc dấu nháy kép. Hoặc chúng ta có thể định nghĩa một inline template với một chuỗi nhiều dòng bằng cách đính kèm mã HTML với Back Tick (`) trong ES 2015. Back Tick cho phép soạn một chuỗi trên nhiều dòng, làm cho mã HTML dễ đọc hơn. Có một số ưu điểm khi định nghĩa một inline-template bằng một trong 2 cách trên, các mẫu (template) được định nghĩa trực tiếp trong các component, cả phần view và phần code đều nằm trong một file. Điều này sẽ dễ dàng để chúng ta kết hợp các ràng buộc dữ liệu trên view với các thuộc tính trong class. Tuy nhiên thì cách này vẫn có nhiều nhược điểm. Khi định nghĩa một HTML trong một chuỗi, hầu hết các công cụ phát triển không cung cấp IntelliSense định dạng tự động và kiểm tra cú pháp. Đặc biệt là khi chúng ta định nghĩa rất nhiều HTML trong template, vẫn đề trở nên khó khăn hơn. Trong trường hợp này, lựa chọn tốt hơn của chúng ta là tạo một file HTML riêng và liên kết Component với nó. Thật may là Angular cung cấp một thuộc tính là “templateUrl” cho phép chúng ta thực hiện cách này. Cuối cùng hãy xem lại những cách mà chúng ta có thể xây dựng một mẫu (template) cho Component.

Inline Template

```
template:  
"<h1>{{pageTitle}}</h1>"
```

Inline Template

```
template: `  
<div>  
  <h1>{{pageTitle}}</h1>  
  <div>  
    My First Component  
  </div>  
</div>  
`
```

Linked Template

```
templateUrl:  
'product-list.component.html'
```

ES 2015
Back Ticks

Bên trong file product-list.component.html chúng ta sẽ theo dõi đoạn code mẫu dưới đây.

<https://codepen.io/quynhlx/pen/WjXELj?editors=1000>

3.3 XÂY DỰNG COMPONENT VÀ SỬ DỤNG NHƯ MỘT DIRECTIVE

Hãy nhớ lại những bước để xây dựng một component mà chúng ta đã trình bày ở chương trước? Chúng ta định nghĩa một class, thêm vào một component decorator để định nghĩa những metadata và chỉ định một mẫu (template). Chúng ta cũng đã import những module chúng ta cần, điều duy nhất thật sự khác với component chúng ta đã tạo ra trong chương trước đó là thuộc tính "template". Ở đây chúng ta dùng templateUrl để xác định vị trí của template thay vì định nghĩa một chuỗi HTML.

product-list.component.ts

```
import { Component } from '@angular/core';  
  
@Component({  
  selector: 'pm-products',  
  templateUrl: 'app/products/product-list.component.html'  
)  
export class ProductListComponent {  
  pageTitle: string = 'Product List';  
}
```

Như vậy **Product List Component** đã có thể sẵn sàng để sử dụng. Cụ thể là chúng ta có thể sử dụng **pm-products** như một chỉ thị (directive). Điều này có nghĩa là chúng ta thể chèn toàn bộ HTML nằm trong file **product-list.component.html** vào bất kỳ một component nào khác, bằng cách dùng selector **<pm-products><pm-products>** như dưới đây.

app.component.ts

```
@Component({  
  selector: 'pm-app',  
  template:  
    <div><h1>{{pageTitle}}</h1>  
    1   <pm-products></pm-products>  
    </div>  
})  
export class AppComponent { }
```

product-list.component.ts

```
@Component({  
  selector: 'pm-products',  
  templateUrl:  
    'app/products/product-list.component.html'  
})  
export class ProductListComponent { }
```

Nhưng chúng ta lại có hàng trăm component trong ứng dụng Angular, làm thế nào để ứng dụng biết nơi để tìm selector này? Hãy nhớ lại thuộc tính Declarations mà chúng ta đã sử dụng để khai báo AppComponent trong AppModule. Chúng ta cũng cần phải khai báo ProductListComponent để AppModule có thể biết được nơi để tìm selector **pm-products**.

3.4 RÀNG BUỘC DỮ LIỆU VỚI INTERPOLATION

Trong Angular, giao tiếp giữa những Class Component và template của nó thường là việc truyền dữ liệu. Chúng ta có thể truyền một giá trị từ Class tới template để hiển thị. Tempate cũng lắng nghe những sự kiện để truyền những hành động hoặc giá trị mà người dùng nhập trở lại Class.



Angular cung cấp một số loại ràng buộc (binding) chúng ta sẽ tìm hiểu tất cả chúng, nhưng trong chương này chúng ta sẽ chỉ nói về Interpolation. Các kỹ thuật binding còn lại sẽ được trình bày trong những bài tiếp theo.

Hãy xem ví dụ dưới đây và chúng ta sẽ nói thêm về Angular Interpolation.

Interpolation

Template	Class
<pre><h1>{{pageTitle}}</h1> {{'Title: ' + pageTitle}} {{2*20+1}} {{'Title: ' + getTitle()}} <h1 innerText={{pageTitle}}></h1></pre>	<pre>export class AppComponent { pageTitle: string = 'Acme Product Management'; getTitle(): string {...}; }</pre>

Ở ví dụ trên, chúng ta có thể nhận thấy ở cả template và Class đều chứa thuộc tính **pageTitle**. Cú pháp `{{ 'Title': + pageTitle}}` trên template khi ứng dụng chạy sẽ cho ra kết quả là **Title: Acme Product Management**. Như vậy đây chính là cách mà chúng ta sẽ truyền giá trị **Acme Product Management** lên template, và đó cũng chính là kĩ thuật Binding with Interpolation.

Ngoài ra chúng ta còn có thể thực hiện một phép toán ngày trong 2 dấu ngoặc kép như ví dụ trên, hay thậm chí là gọi một function `getTitle` từ Class `AppComponent`.

3.5 THÊM LOGIC VỚI DIRECTIVE NGIF.

Chúng ta có thể nghĩ ra một directive như một phần tử hoặc thuộc tính HTML tùy chỉnh để sử dụng tăng "sức mạnh" và mở rộng HTML của chúng ta. Chúng ta có thể xây dựng những directive của riêng mình hoặc sử dụng các directive được xây dựng của Angular.

Chúng ta đã biết cách xây dựng một component và sử dụng nó như một directive. Cụ thể là pm-products directive để hiển thị danh sách những sản phẩm. Tuy nhiên ngoài việc xây dựng những directive riêng, thì chúng ta có thể sử dụng những directive được xây dựng bởi Angular.

Các Angular directive mà chúng ta sẽ xem xét là những directive có liên quan đến kiến trúc khung nhìn. Một structural directive sửa đổi cấu trúc hoặc bố cục của view bằng cách thêm, xóa hoặc thao tác lên các phần tử và con của chúng. Những directive này sẽ giúp chúng ta có thể thêm những đặc trưng như logic như If hay vòng lặp For cho mã HTML.

Angular Built-in Directives

Structural Directives

*ngIf: If logic
*ngFor: For loops

Chú ý dấu hoa thị (*) ở phía trước mỗi directive, nó đánh dấu directive như một structural directive.

Trước tiên, chúng ta sẽ tìm hiểu về directive *ngIf

*ngIf Built-In Directive

```
<div class='table-responsive'>
  <table class='table' *ngIf='products && products.length'>
    <thead> ...
    </thead>
    <tbody> ...
    </tbody>
  </table>
</div>
```

ngIf là một structural directive loại bỏ hoặc tái tạo một phần của cây DOM dựa trên một biểu thức. Nếu biểu thức gán cho ngIf trả về một giá trị false các phần tử và các con của nó sẽ được loại bỏ ra khỏi DOM, còn nếu biểu thức gán cho ngIf trả về một giá trị true các phần tử và con của nó sẽ được thêm vào DOM. Thật đơn giản phải không?

Làm sao để ứng dụng của chúng ta nhận ra ngIf này? ngIf và ngFor là những structural directive được định nghĩa trong BrowserModule, thật may cho chúng ta là BrowserModule đã được chúng ta import trước đó. Và do đó bất kỳ thành phần nào được khai báo bởi AppModule đều có thể sử dụng lệnh ngIf hoặc ngFor.

Tiếp theo, chúng ta hãy xem cách directive *ngFor làm việc.

*ngFor Built-In Directive

```
<tr *ngFor='let product of products'>
  <td></td>
  <td>{{ product.productName }}</td>
  <td>{{ product.productCode }}</td>
  <td>{{ product.releaseDate }}</td>
  <td>{{ product.price }}</td>
  <td>{{ product.starRating }}</td>
</tr>
```

Template
input variable

ngFor lặp lại một phần của cây DOM trong một danh sách lặp lại. Vì vậy, chúng ta xác định một khối HTML cái mà chúng ta muốn hiển thị lặp đi lặp lại để đặt directive ngFor. Ví dụ, chúng ta muốn hiển thị từng sản phẩm trên mỗi hàng của một bảng. Chúng ta sẽ xác định một hàng của bảng và các dữ liệu của hàng đó. Sau đó sẽ lặp đi lặp lại mỗi sản phẩm trong danh sách sản phẩm như ví dụ trên.

Ở ví dụ, chúng ta sử dụng *ngFor='let product of products'. Từ khóa let product sẽ khai báo một biến, biến này sẽ lần lượt mang các giá trị của các phần tử trong danh sách products. Biến product có phạm vi sử dụng trong khối HTML mà chúng ta đặt directive *ngFor.

Một số lưu ý: sẽ có câu hỏi là tại sao chúng ta sử dụng **product of products** mà không dùng **product in products**. Lý do cho điều này là do ES2015. ES2015 định nghĩa cả hai kiểu vòng lặp for. Minh họa dưới đây sẽ giúp các bạn làm sáng tỏ điều này.

for...of vs for...in

for...of

- Iterates over iterable objects, such as an array.
- Result: di, boo, punkeye

```
let nicknames= ['di', 'boo', 'punkeye'];

for (let nickname of nicknames) {
  console.log(nickname);
}
```

for...in

- Iterates over the properties of an object.
- Result: 0, 1, 2

```
let nicknames= ['di', 'boo', 'punkeye'];

for (let nickname in nicknames) {
  console.log(nickname);
}
```

Như vậy, tùy vào từng trường hợp mà chúng ta sẽ sử dụng kiểu vòng lặp for nào cho phù hợp. Nếu muốn sử dụng chỉ mục index, hay sử dụng for in, nếu muốn sử dụng một đối tượng hãy dùng for of.

3.6 TỔNG KẾT

Template



Inline template

- Dành cho mẫu ngắn
- Xác định bằng thuộc tính **template**
- Sử dụng ES 2015 back ticks cho nhiều dòng

Linked template

- Dành cho mẫu template dài
- Xác định bằng thuộc tính **templateUrl**
- Định rõ đường dẫn tới tệp HTML

Checklist: Component as a Directive

product-list.component.ts

```
@Component({
  selector: 'pm-products',
  templateUrl:
    'app/products/product-list.component.html'
})
export class ProductListComponent { }
```

app.component.ts

```
1 @Component({
  selector: 'pm-app',
  template:
    `<div><h1>{{pageTitle}}</h1>
     <pm-products></pm-products>
    </div>`
})
export class AppComponent { }
```

app.module.ts

```
2 @NgModule({
  imports: [ BrowserModule ],
  declarations: [
    AppComponent,
    ProductListComponent ],
  bootstrap: [ AppComponent ]
})
export class AppModule { }
```

Interpolation



Binding một chiều

- Từ thuộc tính của class tới một thuộc tính element.

Xác định với hai dấu ngoặc nhọn {{}}

- Chứa một template expression
- Không cần dấu nháy kép

Structural Directives



*ngIf and *ngFor

- Tiễn tố với một dấu hoa thị.
- Gán cho một biểu thức chuỗi trong dấu nháy.

*ngIf

- Biểu thức là có giá trị true hoặc false

*ngFor

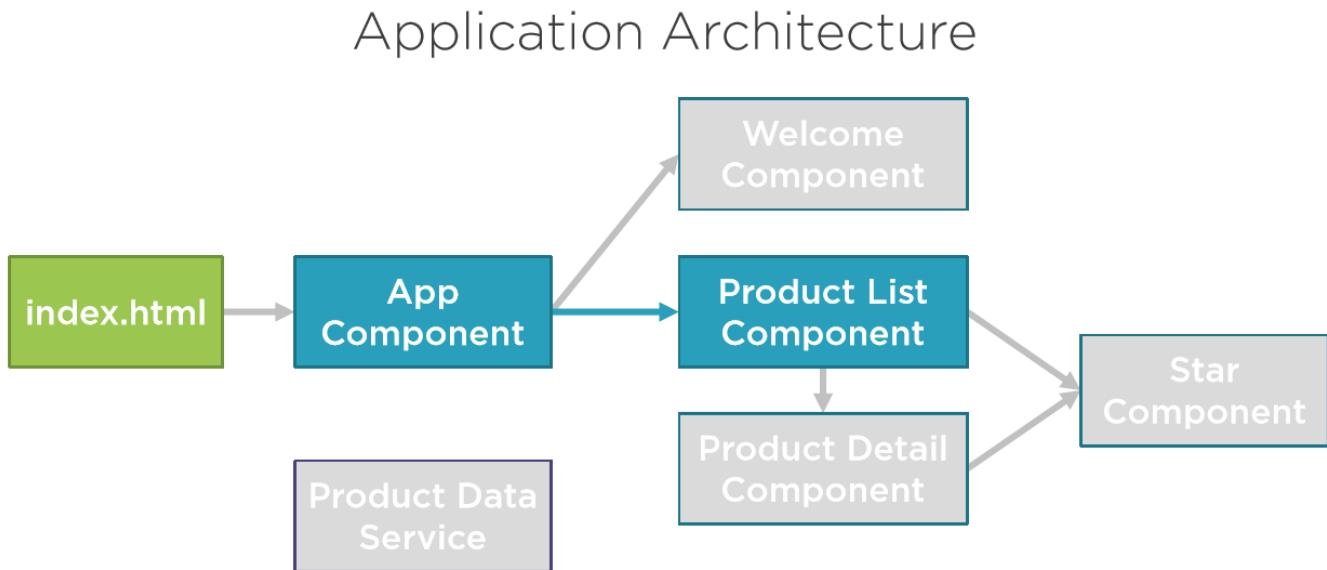
- Định nghĩa biến cục bộ với let
- Sử dụng 'of': let product of products

4. DATA BINDING & PIPES

4.1 GIỚI THIỆU

Có nhiều thứ cần liên kết dữ liệu hơn là chỉ hiển thị thuộc tính của component. Bài học hôm nay chúng ta sẽ khám phá nhiều tính năng liên kết dữ liệu và biến đổi dữ liệu với pipes (đường ống). Để cung cấp một trải nghiệm người dùng tuyệt vời, chúng ta muốn ràng buộc phần tử DOM vào thuộc tính của component để component có thể thay đổi giao diện khi cần thiết. Chúng ta có thể sử dụng liên kết để thay đổi các phần tử màu sắc hoặc style dựa trên các giá trị dữ liệu, như việc chúng ta có thể cập nhật font size dựa trên user preferences hoặc thiết lập một hình ảnh từ một trường trên cơ sở dữ liệu. Hơn thế nữa, chúng ta muốn thông báo về các hành động của người dùng lên DOM để component có thể phản hồi lại. Ví dụ, chúng ta nhấp chuột lên một nút để ẩn hoặc hiển thị hình ảnh. Và đôi khi, chúng ta muốn liên kết dữ liệu sẽ là hai chiều để có thể nhận được những sự kiện thay đổi của những thuộc tính mà chúng ta hiển thị.

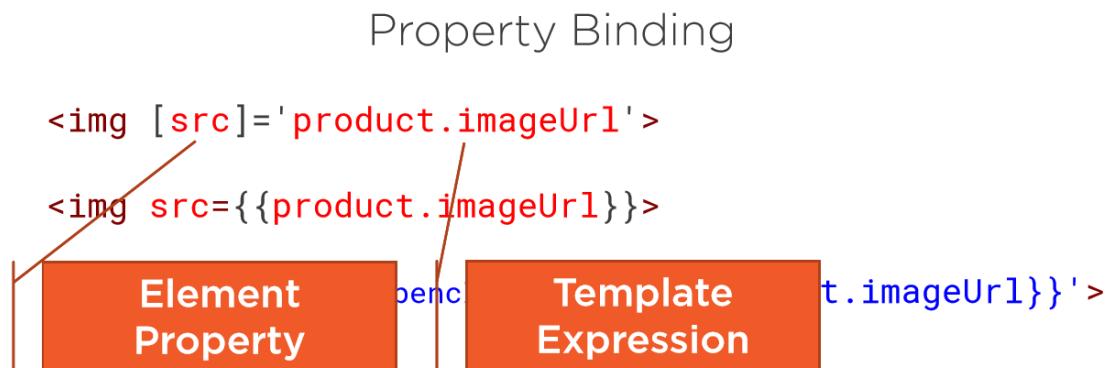
Ở chương này, chúng ta sẽ sử dụng properties binding để thiết lập các thuộc tính phần tử HTML trong DOM. Chúng ta sẽ đi sâu vào việc làm thế nào để xử lý các sự kiện người dùng như khi ta click vào một nút bấm. Và làm thế nào để xử lý với binding 2 chiều. Cuối cùng chúng ta sẽ khám phá làm thế nào để chuyển đổi dữ liệu ràng buộc với pipes.



Chúng ta hãy xem lại kiến trúc ứng dụng một lần nữa, chúng ta đã thực hiện một phần của Product List Component, tuy nhiên component hiện tại mới chỉ hiển thị dữ liệu sản phẩm, mà chưa có bất cứ hành động hay tương tác nào lên những sản phẩm này. Ở bài học này, chúng ta sẽ sử dụng những đặc tính binding để thêm các tương tác vào Product List Component.

4.2 PROPERTY BINDING

Property binding (liên kết thuộc tính) cho phép chúng ta thiết lập property của một element với giá trị của một Template Expression



Để so sánh sự khác nhau giữa property binding và interpolation, chúng ta hãy xem đoạn 2 đoạn code trên. Hãy lưu ý, đoạn code đầu tiên thuộc tính của element được bao quanh bởi dấu ngoặc vuông nhưng template expression thì lại không có dấu ngoặc nhọn mà được bao bởi 2 dấu nháy đơn. Nhưng cũng giống như interpolation, property binding là liên kết một chiều từ thuộc tính của class tới thuộc tính của element. Nó cho phép chúng ta kiểm soát cây DOM từ class component.

Cả property binding và interpolation đều cho ra kết quả giống nhau, vậy dùng cái nào thì tốt hơn? Các hướng dẫn thông thường thì đều dùng property binding, tuy nhiên nếu template expression của bạn là một phần của một biểu thức lớn như ví dụ dưới đây, bạn cần phải dùng interpolation.

```
<img src='http://openclipart.org/{{product.imageUrl}}'>
```

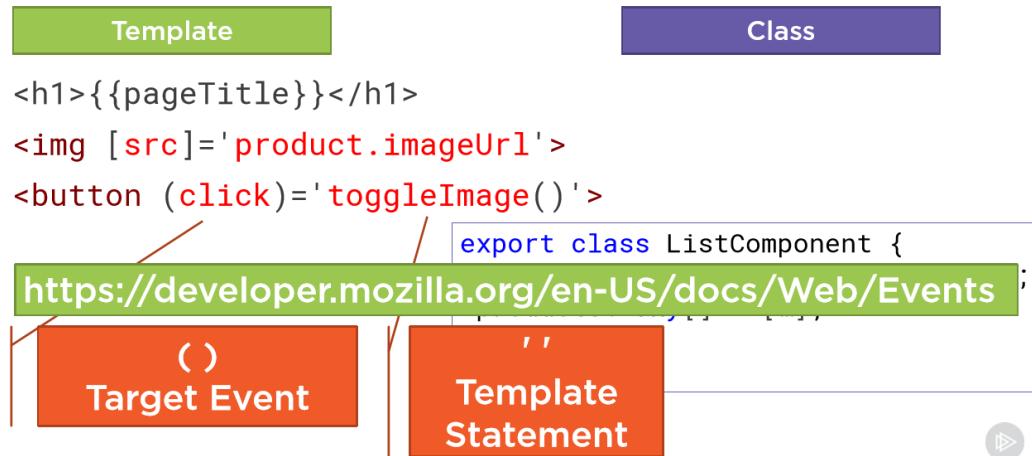
Bây giờ hãy theo dõi code ứng dụng mẫu của chúng ta để có cái nhìn tổng quát.

4.3 XỬ LÝ SỰ KIỆN VỚI EVENT BINDING

Cho đến bây giờ, tất cả các dữ liệu liên kết của chúng ta đều là một chiều từ component class tới thuộc tính element, nhưng có những lúc chúng ta cần gửi thông tin từ người dùng phản hồi về lại component class, ví dụ như khi người dùng thực hiện thao tác click chuột vào một button.

Một component để có thể lắng nghe các hành động của người dùng bằng cách sử dụng event binding như ví dụ dưới đây.

Event Binding



Chúng ta cũng nhận thấy, cú sử dụng event binding có phần giống như property binding. Trong ví dụ này, component lắng nghe sự kiện click trên button. Tên của sự kiện được bao bởi dấu ngoặc đơn còn được gọi là Target Event. Phía bên phải sau dấu "=" là template statement. Nó thường là tên của một phương thức trong component class theo sau đó là dấu (). Nếu như có sự kiện xảy ra, template stagement này sẽ thực hiện gọi phương thức đã được định nghĩa trong component.

4.4 XỬ LÝ INPUT VỚI TWO-WAY BINDING

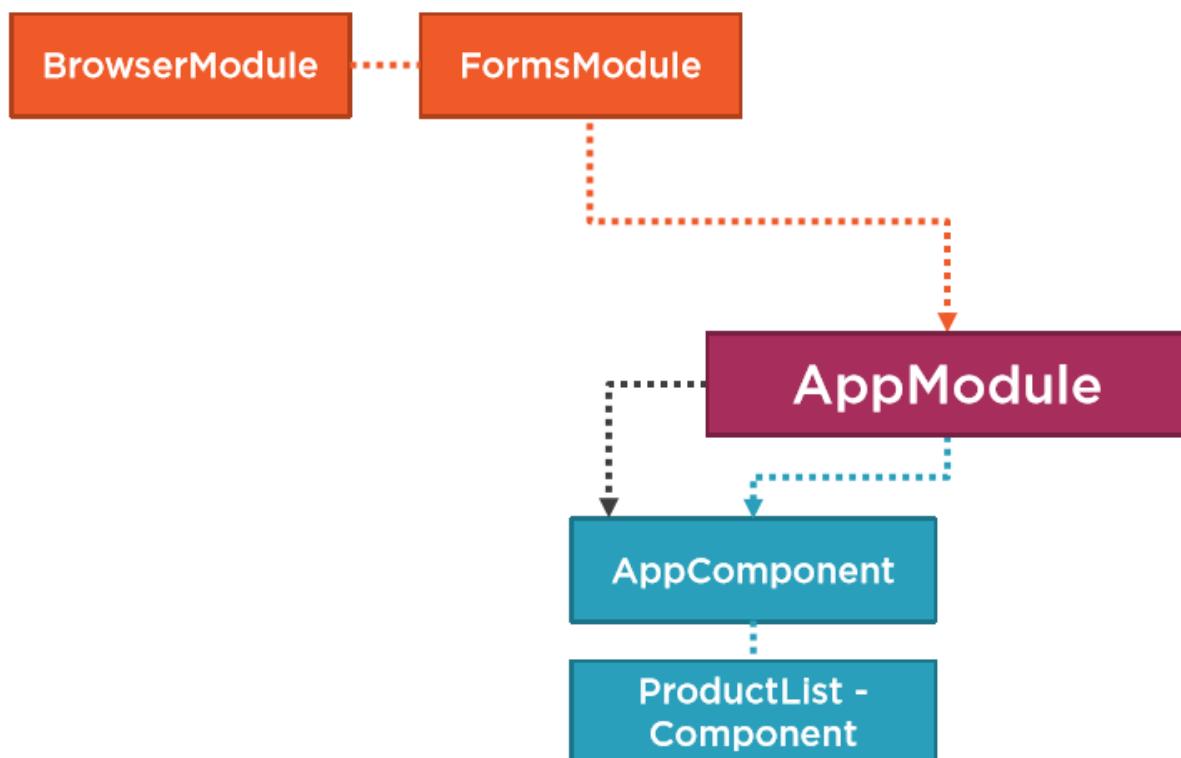
Khi làm việc với những phần tử HTML cho phép nhập liệu như <input>, chúng ta thường muốn hiển thị một thuộc tính trong một component class lên template và cập nhật thuộc tính đó mỗi khi người dùng thay đổi nó. Quá trình này đòi hỏi sự ràng buộc hai chiều (two-way binding). Để chỉ định two-way binding trong Angular chúng ta sử dụng directive **ngModel**. Chúng ta thêm vào ngModel dấu ngoặc vuông để cho thuộc tính đó liên kết từ thuộc tính của class đến input element, và thêm vào dấu ngoặc đơn để chỉ ra thuộc tính sẽ có một sự kiện thông báo cho thuộc tính của class mỗi khi người dùng nhập dữ liệu.

Two-way Binding



Ở ví dụ này, mỗi khi người dùng thay đổi giá trị trên input, lập tức giá trị của thuộc tính bên trong classListComponent cũng sẽ thay đổi giá trị theo, và ngược lại, khi chúng ta thay đổi giá trị thuộc tính listFilter thì giao diện người dùng cũng sẽ ngay lập tức cập nhật giá trị tương ứng.

Cũng như những directive mà chúng ta đã tìm hiểu ở các bài trước, chúng ta cần phải import module, cái mà đã định nghĩa sẵn những directive này cho chúng ta. Đối với directive ngModel, nó không nằm trong BrowserModule mà lại nằm trong FormsModule. Nên tại AppModule chúng ta cần phải import FormsModule vào để các component thuộc AppModule có thể sử dụng được directive ngModel. Chúng ta có thể xem minh họa dưới đây.



..... Imports

..... Exports

..... Declarations

..... Providers

..... Bootstrap

4.5 BIẾN ĐỔI DỮ LIỆU VỚI PIPES

Với Angular data binding, việc hiển thị dữ liệu trở nên thật dễ dàng. Chỉ cần bind một thuộc tính của element với một thuộc tính của class là xong. Nhưng, không phải lúc nào cũng vậy. Thỉnh thoảng dữ liệu của chúng ta không có định dạng phù hợp để hiển thị. Đó là lúc pipes trở nên có ích. Pipes sẽ biến đổi thuộc tính ràng buộc trước khi chúng ta hiển thị do đó chúng ta có thể chỉnh sửa giá trị của thuộc tính để chúng thân thiện và phù hợp hơn với người dùng. Angular cũng cung cấp một số pipes được xây dựng sẵn cho việc định dạng dữ liệu như date, number, decimal, percent, currency, uppercase, lowercase, v.v... Angular cũng cung cấp một vài pipes để làm việc với đối tượng như JSON pipe để hiển thị nội dung của một đối tượng như một chuỗi JSON, điều này sẽ giúp ích rất nhiều khi debugging.

Ngoài ra chúng ta cũng có thể xây dựng những custom pipes, chúng ta sẽ được tìm hiểu nó trong bài học tiếp theo. Ở bài học này, chúng ta hãy bắt đầu với những ví dụ đơn giản.

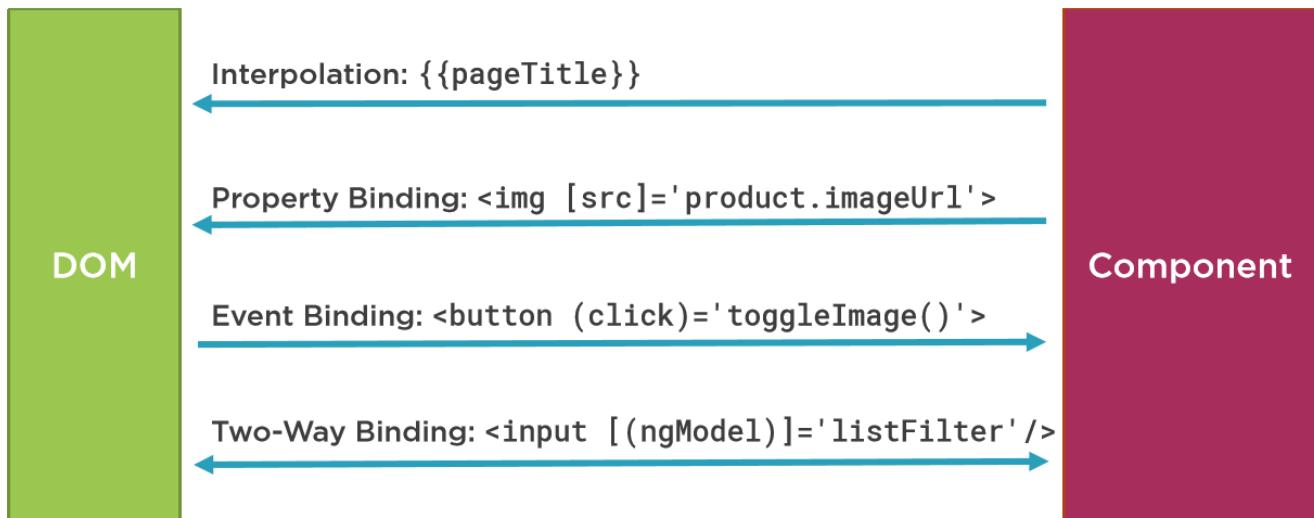
Pipe Examples

```
 {{ product.productCode | lowercase }}  
  
 <img [src] ='product.imageUrl'  
       [title] ='product.productName | uppercase'>  
  
 {{ product.price | currency | lowercase }}  
  
 {{ product.price | currency:'USD':true:'1.2-2' }}
```

Ở ví dụ này, chúng ta sử dụng pipe để biến đổi mã sản phẩm, giá tiền thành chữ thường, tên sản phẩm là chữ hoa. Hãy chú ý đến cú pháp để áp dụng một pipe. Ngoài ra chúng cũng có thể sử dụng nhiều pipe lồng nhau để biến đổi dữ liệu như mong muốn, hoặc tự định dạng dữ liệu theo một kiểu mà ta mong muốn.

4.6 TỔNG KẾT

Data Binding



Checklist: ngModel

product-list.component.html

```
<div class='col-md-4'>
  <input type='text'
    [(ngModel)]=listFilter' />
</div>
```

app.module.ts

```
@NgModule({
  imports: [
    BrowserModule,
    FormsModule ],
  declarations: [
    AppComponent,
    ProductListComponent ],
  bootstrap: [ AppComponent ]
})
export class AppModule { }
```

Checklist: Pipes



Pipe character |

Pipe name

Pipe parameters

- Separated with colons

Example

- `{{ product.price | currency:'USD':true:'1.2-2' }}`

5. COMPONENT CHUYÊN SÂU

5.1 GIỚI THIỆU

Khi xây dựng những component “sạch”. Chúng ta muốn chắc chắn mọi thứ trong component sẽ là strongly typed, tất cả những styles đều được đóng gói, chúng ta đáp ứng những sự kiện vòng đời của component một cách thích hợp và chúng ta chuyển đổi dữ liệu để chúng thân thiện với người dùng. Bài học hôm nay, chúng ta sẽ học một vài cách cải tiến các component của chúng ta. Những components là chìa khoá để xây dựng ứng dụng của chúng ta. Một component “sạch”, mạnh, và kiên cố sẽ giúp ứng dụng của chúng ta tốt hơn. Vậy chúng ta cần làm gì để cải tiến component của chúng ta? Chúng ta sẽ lần lượt đi qua các mục dưới đây để trả lời câu hỏi này.

1. Định nghĩa một Interface
2. Đóng gói Component Styles
3. Sử dụng Lifecycle Hooks
4. Xây dựng Custom Pipe

5.2 ĐỊNH NGHĨA MỘT INTERFACE

Một trong những lợi ích sử dụng TypeScript là strong typing. Mọi thuộc tính đều xác định một kiểu dữ liệu, mỗi phương thức cũng đều trả về một kiểu dữ liệu, những tham số truyền vào phương thức cũng vậy. Việc áp dụng strong typing giúp giảm thiểu các lỗi thông qua việc kiểm tra cú pháp.

```
export class ProductListComponent {  
    pageTitle: string = 'Product List';  
    showImage: boolean = false;  
    listFilter: string = 'cart'; ←  
    message: string;    Các thuộc tính đều có kiểu dữ liệu  
  
    products: any[] = [...];    Một số trường hợp  
                                không có kiểu dữ liệu xác định  
  
    toggleImage(): void {    phương thức cũng trả về  
        this.showImage = !this.showImage;  
    }  
  
    onRatingClicked(message: string): void {  
        this.message = message;    tham số truyền xác định  
                                kiểu dữ liệu  
    }  
}
```

Trong ví dụ trên, chúng ta thấy products là một trường hợp đặc biệt sử dụng kiểu dữ liệu any (bất kỳ). Điều này phủ nhận lợi ích của strong typing. Do đó để xác định một kiểu dữ liệu tùy chỉnh, chúng ta cần định nghĩa một interface.

- Một interface là một đặc tả xác định những thuộc tính và phương thức có liên quan.
- Một class cam kết hỗ trợ các đặc tả bằng cách implementing các interface.
- Chúng ta sử dụng interface như một kiểu dữ liệu.

Mặc dù ES5 và ES 2015 không hỗ trợ interface nhưng TypeScript đã làm điều đó, nên bạn không cần quá bận tâm. Hơn nữa interface sẽ không xuất hiện trong đoạn mã Javascript sau khi build. Điều này cũng có nghĩa rằng interface chỉ phục vụ cho thời điểm phát triển. Mục đích của chúng là cung cấp strong typing, hỗ trợ công cụ tốt hơn khi xây dựng, gỡ rối, và bảo trì code của chúng ta.

Dưới đây là một ví dụ về TypeScript Interface.

Interface Is a Specification

```
export interface IProduct {  
    productId: number;  
    productName: string;  
    productCode: string;  
    releaseDate: Date;  
    price: number;  
    description: string;  
    starRating: number;  
    imageUrl: string;  
    calculateDiscount(percent: number): number;  
}
```



Chúng ta định nghĩa một interface bằng từ khoá "interface", theo sau đó là tên của interface. Theo quy ước đặt tên, interface có tiền tố là "I" ở ví dụ của chúng ta là IProduct, điều này rất nhiều nhà phát triển thường hay bỏ qua. Từ khoá export ở đầu tiên để khai báo interface này đã sẵn sàng sử dụng ở bất cứ nơi nào trong ứng dụng.

Để sử dụng interface này như một kiểu dữ liệu, chúng ta import interface này, sau đó có thể sử dụng interface như một kiểu dữ liệu. Trong sẽ dễ dàng như đoạn ví dụ dưới đây.

Using an Interface as a Data Type

```
import { IProduct } from './product';  
  
export class ProductListComponent {  
    pageTitle: string = 'Product List';  
    showImage: boolean = false;  
    listFilter: string = 'cart';  
  
    products: IProduct[] = [...];  
  
    toggleImage(): void {  
        this.showImage = !this.showImage;  
    }  
}
```

5.3 ĐÓNG GÓI COMPONENT STYLES

Khi chúng ta xây dựng template cho một component, thỉnh thoảng chúng ta sẽ cần những styles duy nhất cho component này. Ví dụ, chúng ta xây dựng một sidebar navigation (thanh điều hướng) component, chúng ta có thể muốn một số styles đặc biệt cho thẻ li hay div, sẽ cần một cách để mang những styles chỉ sử dụng cho component này.

Có một cách là chúng ta sẽ định nghĩa trực tiếp những style này vào trong đoạn mã HTML của template. Nhưng như vậy thì sẽ khó để xem, sử dụng lại hay bảo trì những đoạn code chứa style này. Một lựa chọn khác là định nghĩa một file css bên ngoài, điều này sẽ dễ bảo trì code, nhưng chúng ta sẽ phải chắc chắn rằng file css sẽ phải được liên kết ở trong file index.html. Và như vậy lại khó khăn trong vấn đề sử dụng lại component. Có một cách tốt hơn cả giúp chúng ta giải quyết vấn đề này, component decorator có những thuộc tính để đóng gói những style lại như một phần của component. Những thuộc tính này là "styles" và "styleUrls".

styles

```
@Component({
  selector: 'pm-products',
  templateUrl: 'app/products/product-list.component.html',
  styles: ['thead {color: #337AB7;}']})
```

styleUrls

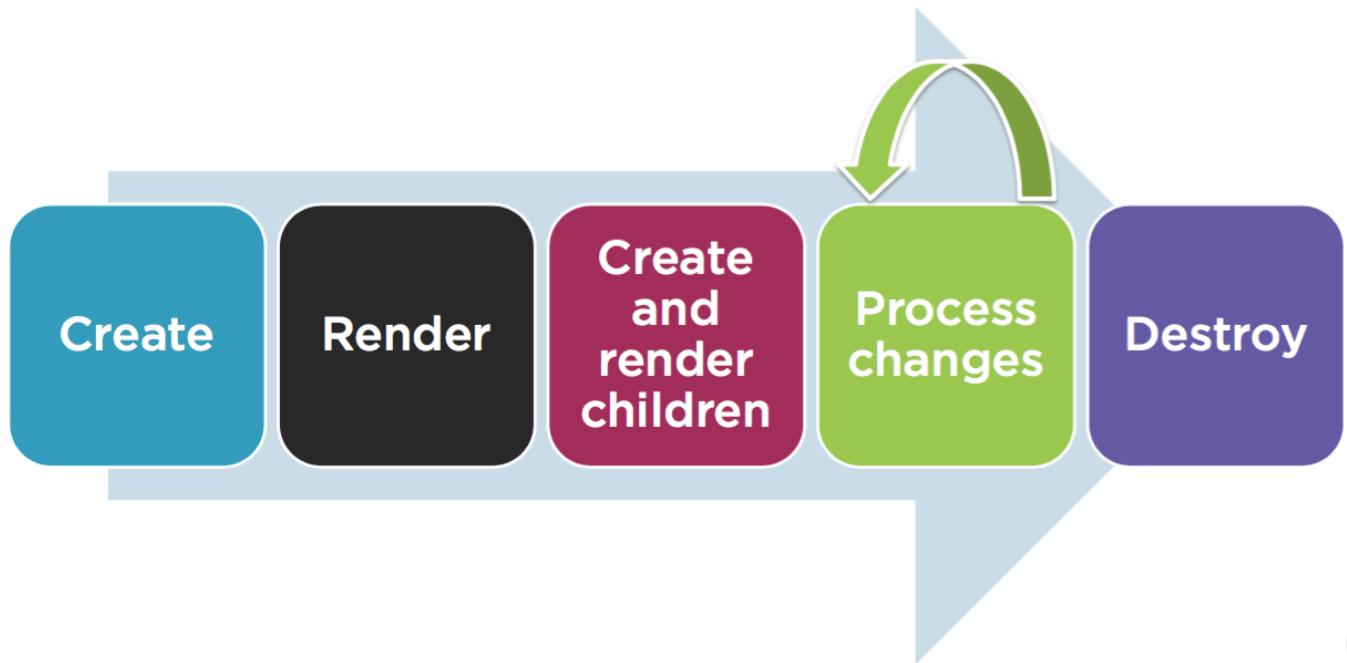
```
@Component({
  selector: 'pm-products',
  templateUrl: 'app/products/product-list.component.html',
  styleUrls: ['app/products/product-list.component.css']})
```

Với styles và styleUrls, chúng đều cho phép khai báo một mảng những phần tử. Do đó chúng ta có thể thêm nhiều hơn một style hay một file stylesheet bên ngoài.

5.4 SỬ DỤNG LIFECYCLE HOOKS

Một component có một vòng đời, được quản lý bởi Angular. Angular tạo ra một component, render nó, khởi tạo và render những thành phần con, tiến hành thay đổi khi thuộc tính liên kết của component thay đổi, và cuối cùng phá huỷ chúng trước khi xoá chúng khỏi DOM.

Component Lifecycle



Angular cung cấp một bộ quản lý vòng đời, chúng ta có thể sử dụng để can thiệp vào vòng đời này và thực hiện những hành động cần thiết. Vì đây là khóa học cơ bản, chúng ta chỉ giới hạn tập trung vào ba vòng đời đơn giản nhất.

1. **OnInit:** Thực hiện khởi tạo những thứ bên trong component sau khi Angular đã khởi tạo những thuộc tính liên kết dữ liệu. Đây là nơi để lấy dữ liệu cho template từ một back-end service, chúng ta sẽ làm rõ điều này trong chương tiếp theo.
2. **OnChanges:** Thực hiện bất kỳ hành động nào sau khi Angular thiết lập những thuộc tính liên kết input.
3. **OnDestroy:** Thực hiện việc dọn dẹp trước khi Angular sẽ hủy component.

Để sử dụng một Lifecycle Hook, chúng ta “implements” Lifecycle Hook interface mà chúng ta muốn. Chúng ta đã nói về interface ở trong chương trước đây. Angular cung cấp nhiều interfaces xây dựng sẵn mà chúng ta có thể implement, bao gồm cả những interface ứng với mỗi lifecycle hook. Dưới đây là một ví dụ mà chúng ta implement OnInit.

Using a Lifecycle Hook

```
2 import { Component, OnInit } from '@angular/core';
1 export class ProductListComponent
    implements OnInit {
  pageTitle: string = 'Product List';
  showImage: boolean = false;
  listFilter: string = 'cart';
  products: IProduct[] = [...];
3   ngOnInit(): void {
      console.log('In OnInit');
    }
}
```

Đầu tiên chúng ta sẽ implements OnInit, tuy nhiên để sử dụng interface OnInit chúng ta cần import nó từ @angular/core. Cuối cùng chúng ta sẽ thực hiện những gì cần thiết trong hàm ngOnInit. Mỗi một interface lifecycle hook sẽ có những phương thức khác nhau khi chúng ta implement, trong ví dụ này phương thức chúng ta implement là "ngOnInit".

Hãy nhớ lại một chút về những gì chúng ta thảo luận về interface trong phần trước rằng interface thì không được hỗ trợ bởi ES5 và ES2015, đây là những đặc trưng của TypeScript. Điều này có nghĩa là kết quả của những dòng code Typescript của chúng ta cũng trả về là javascript. Vì vậy chúng ta không thực sự cần phải implement những interface để có thể sử dụng lifecycle hook. Đơn giản chúng ta có thể viết hook function (như ngOnInit). Tuy nhiên, sẽ tốt hơn nếu chúng ta implement như những gì Angular đã khuyến cáo, bởi như vậy chúng ta sẽ cung cấp tối ưu cho các công cụ editor, và cũng giúp code của chúng ta rõ ràng hơn.

5.5 XÂY DỰNG NHỮNG CUSTOM PIPE.

Như chúng ta đã thấy trong chương trước, chúng ta sử dụng những pipes để biến đổi những thuộc tính ràng buộc trước khi hiển thị chúng trên một view. Angular có sẵn pipe đó được xây dựng để biến đổi một giá trị hoặc một danh sách dữ liệu lặp lại. Đó là những gì chúng ta bàn ở chương trước. Trong chương này, chúng ta muốn xây dựng một custom pipe của riêng chúng ta. Đoạn code xây dựng một custom pipe dưới đây có thể sẽ trông rất quen thuộc.

Building a Custom Pipe

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

@Pipe({
  name: 'productFilter'
})
export class ProductFilterPipe
  implements PipeTransform {

  transform(value: IProduct[],
            filterBy: string): IProduct[] {
  }
}
```

Ở ví dụ này, chúng ta sẽ xây dựng một custom pipe có nhiệm vụ lọc những sản phẩm (value: IProduct[]) theo filterBy và trả về những sản phẩm có cùng kiểu dữ liệu IProduct.

Chúng ta cần lưu ý, trước class chúng ta thêm một pipe decorator để định nghĩa class này là một pipe, thuộc tính name trong decorator cũng chính là tên của pipe sẽ được dùng. Dĩ nhiên chúng ta cũng không quên import Pipe, và PipeTransform từ @angular/core để class có thể implement chúng.

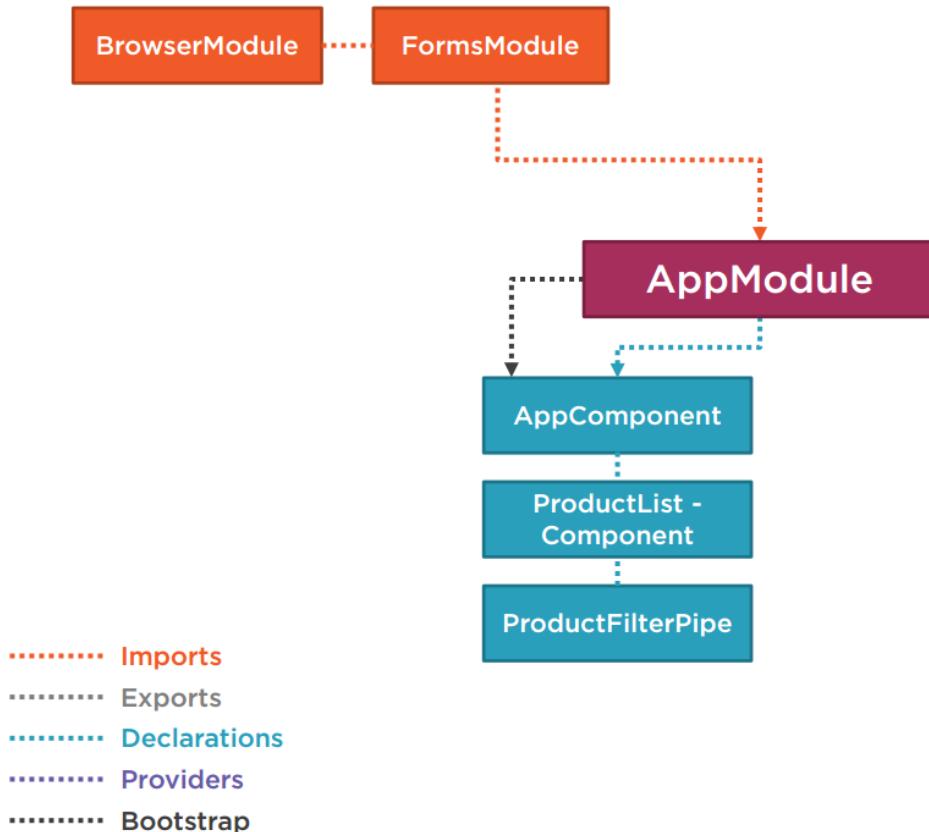
Interface PipeTransform sẽ yêu cầu class phải implement phương thức transform. Chúng ta sẽ thực hiện chuyển đổi dữ liệu trong hàm này.

Sau khi đã định nghĩa một custom pipe, để sử dụng nó chúng ta làm tương tự như những built-in pipes mà Angular cung cấp. Ở template chúng ta sẽ sử dụng custom pipe như dưới đây.

Template

```
<tr *ngFor ='let product of products | productFilter: listFilter'>
```

Bây giờ hãy nhìn lại một chút toàn bộ kiến trúc mà ứng dụng chúng ta đang xây dựng. Chúng ta vừa tạo ra một ProductFilterPipe, do đó để có thể sử dụng nó trong AppModule, ngoài những bước trên chúng ta cần phải khai báo cho AppModule biết về Custom Pipe mới này.



Cách mà chúng ta sẽ khai báo cho AppModule như đoạn code dưới đây.

Module
<pre> @ NgModule({ imports: [BrowserModule, FormsModule], declarations: [AppComponent, ProductListComponent, ProductFilterPipe], bootstrap: [AppComponent] }) export class AppModule { } </pre>

5.6 TỔNG KẾT

Checklist: Interfaces



Defines custom types

Creating interfaces:

- `interface` keyword
- export it

Implementing interfaces:

- `implements` keyword & interface name
- Write code for each property & method

Checklist: Encapsulating Styles



styles property

- Specify an array of style strings

styleUrls property

- Specify an array of stylesheet paths

Checklist: Using Lifecycle Hooks



- Import the lifecycle hook interface
- Implement the lifecycle hook interface
- Write code for the hook method

Checklist: Building a Custom Pipe



- Import Pipe and PipeTransform
- Create a class that implements PipeTransform
 - export the class
- Write code for the Transform method
- Decorate the class with the Pipe decorator

Checklist: Using a Custom Pipe



Import the custom pipe

Add the pipe to the declarations array of an Angular module

Any template associated with a component that is also declared in that Angular module can use that pipe

Use the Pipe in the template

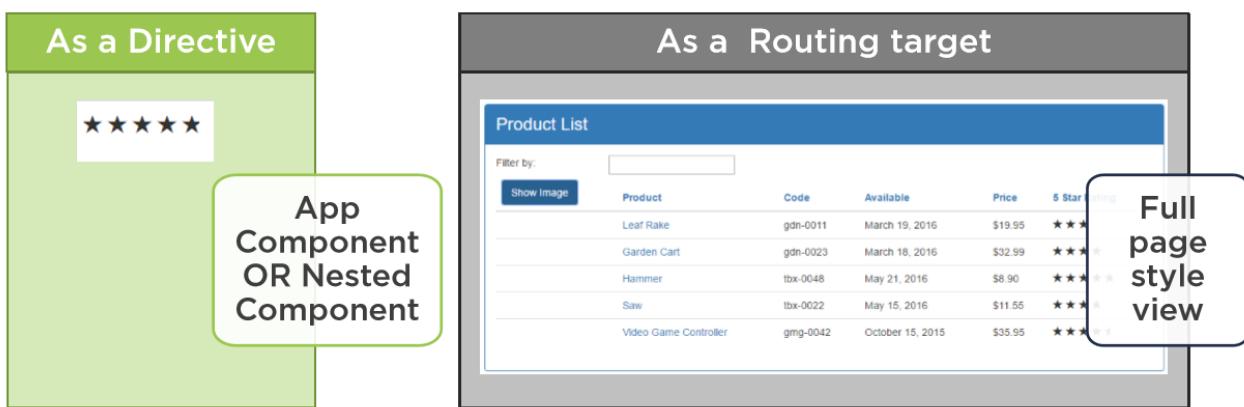
- Pipe character
- Pipe name
- Pipe arguments (separated with colons)

6. XÂY DỰNG NHỮNG COMPONENT LỒNG NHAU

6.1 GIỚI THIỆU

Thiết kế giao diện người dùng của chúng ta đôi khi quá phức tạp, và bắt chúng ta phải nghĩ đến giải pháp chia nhỏ chúng thành nhiều components nhỏ hơn. Hoặc có một số phần được sử dụng lại ở nhiều nơi. Trong chương này, chúng ta sẽ xem cách làm thế nào để xây dựng những component thiết kế để có thể lồng vào trong component khác và chúng ta cũng khám phá cách thiết lập giao tiếp giữa component lồng bên trong và component cha của nó. Bởi vì mỗi một thành phần đều được đóng gói, chúng ta chỉ ra những input và những output cụ thể để component có thể giao tiếp với nhau.

Có hai cách để sử dụng một component và hiển thị template của nó:

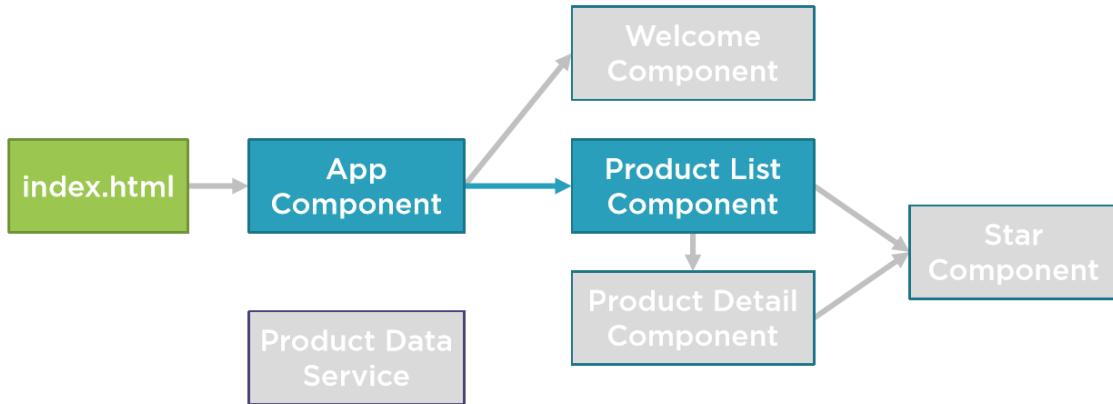


1. Chúng ta sử dụng component như một directive, chúng ta đã tìm hiểu về kỹ thuật này.
2. Ngoài ra, chúng ta có thể sử dụng một component như một địa chỉ định tuyến, chúng ta sẽ tìm hiểu thêm về kỹ thuật này ở chương sau.

Cả hai cách này đều có điểm chung là chia nhỏ một view (khung nhìn) thành nhiều component nhỏ hơn để dễ quản lý. Tuy nhiên vì chúng ta chưa tìm hiểu về routing nên trong giới hạn chương này, chúng ta sẽ tìm hiểu về cách giao tiếp component lồng nhau (Nested Components) như một directive.

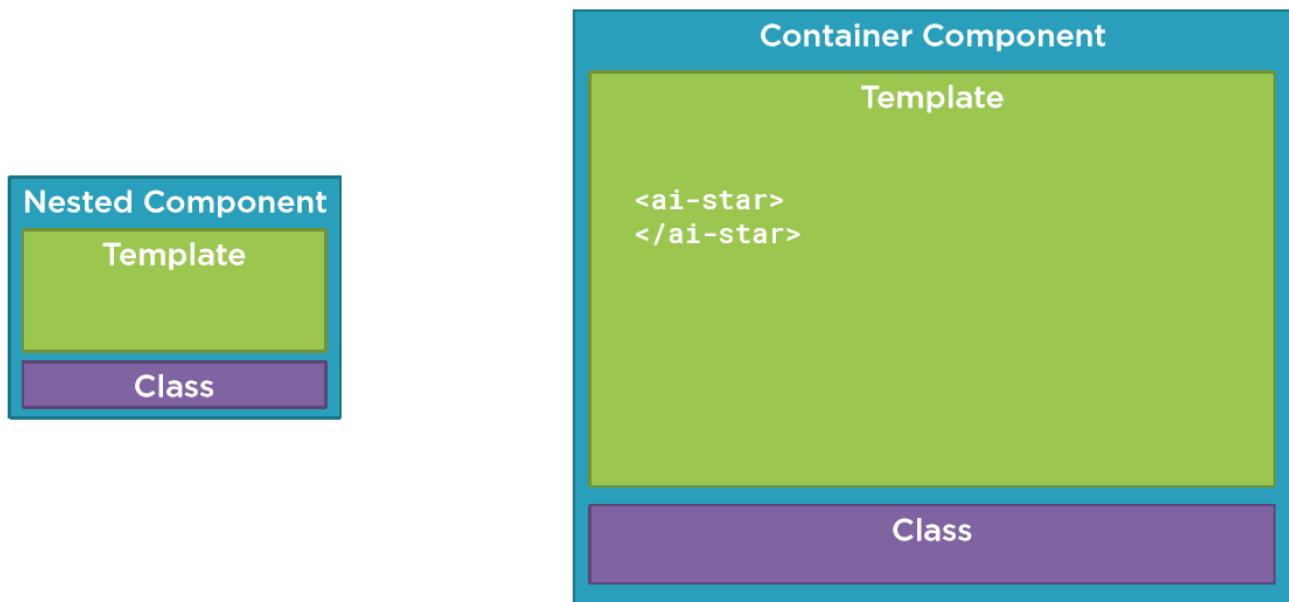
Hãy cùng nhau xem lại kiến trúc của ứng dụng trước khi chúng ta đi tiếp.

Application Architecture

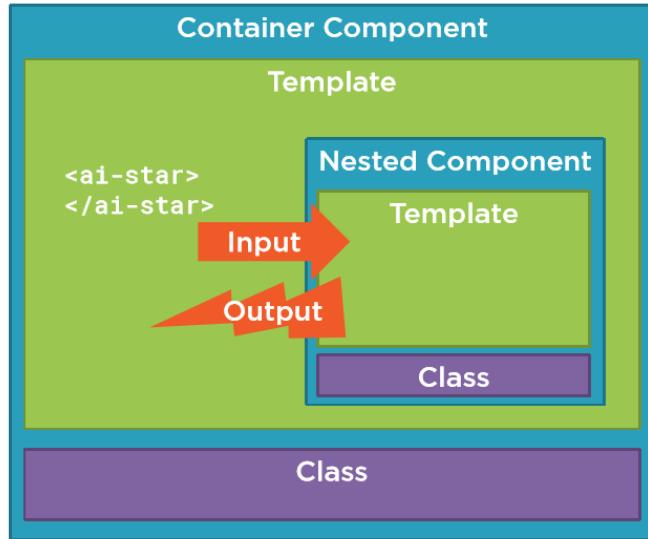


6.2 XÂY DỰNG MỘT COMPONENT LỒNG NHAU (NESTED COMPONENT)

Dưới đây là một hình ảnh đại diện cho một component lồng nhau, và một component sẽ chứa nó.



Khi chúng ta xây dựng một ứng dụng tương tác, nested component thường cần giao tiếp với component cha của nó. Nested component nhận thông tin từ cha nó thông qua thuộc tính input và nested component trả ra những thông tin về lại thông tin về cha của nó bằng cách bắn ra các sự kiện.



Trong ứng dụng mẫu mà chúng ta đang xây dựng, chúng ta muốn thay đổi hiển thị của 5 start rating từ những con số thành các ngôi sao. Việc hiển thị số đánh giá bằng cách sử dụng biểu diễn trực quan như các ngôi sao làm cho người dùng giải thích ý nghĩa của các con số nhanh và dễ dàng hơn.

Trước

Product List					
Show Image	Product	Code	Available	Price	5 Star Rating
	Leaf Rake	GDN-0011	March 19, 2016	\$19.95	3.2
	Garden Cart	GDN-0023	March 18, 2016	\$32.99	4.2
	Hammer	TBX-0048	May 21, 2016	\$8.9	4.8
	Saw	TBX-0022	May 15, 2016	\$11.55	3.7
	Video Game Controller	GMG-0042	October 15, 2015	\$35.95	4.6

Sau

Product List

Show Image	Product	Code	Available	Price	5 Star Rating
	Leaf Rake	GDN-0011	Mar 19, 2016	\$19.95	★★★☆
	Garden Cart	GDN-0023	Mar 18, 2016	\$32.99	★★★★☆
	Hammer	TBX-0048	May 21, 2016	\$8.99	★★★★★
	Saw	TBX-0022	May 15, 2016	\$11.55	★★★★☆
	Video Game Controller	GMG-0042	Oct 15, 2015	\$35.95	★★★★★

Đây là nested component, cái mà chúng ta sẽ xây dựng trong bài học này. Chúng ta sẽ gọi component này là Start Component.

Để Start Component hiển thị chính xác số sao đánh giá, thành phần chứa component này sẽ phải cung cấp rating number tới Start Component thông qua một input. Khi người dùng click lên những ngôi sao, chúng ta muốn phát ra một event để thông báo cho thành phần chứa Start Component biết chúng ta đã thay đổi start rating này. Ngay bây giờ hãy bắt tay vào xây dựng Start Component.

star.component.ts

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

@Component({
  selector: 'ai-star',
  templateUrl: 'app/shared/star.component.html',
  styleUrls: ['app/shared/star.component.css']
})
export class StarComponent implements OnChanges {
  rating: 4;
  starWidth: number;

  ngOnChanges(): void {
    this.starWidth = this.rating * 86 / 5;
  }
}
```

star.component.html

```
<div class="crop"
  [style.width.px]="starWidth"
```

```
[title]="rating"
<div style="width: 86px">
  <span class="glyphicon glyphicon-star"></span>
  <span class="glyphicon glyphicon-star"></span>
  <span class="glyphicon glyphicon-star"></span>
  <span class="glyphicon glyphicon-star"></span>
  <span class="glyphicon glyphicon-star"></span>
</div>
</div>
```

6.3 SỬ DỤNG MỘT NESTED COMPONENT

product-list.component.ts

```
@Component({
  selector: 'pm-products',
  templateUrl: 'product-list.component.html'
})
export class ProductListComponent { }
```

star.component.ts

```
@Component({
  selector: 'ai-star',
  templateUrl: 'star.component.html'
})
export class StarComponent {
  rating: number;
  starWidth: number;
}
```

product-list.component.html

```
<td>
  <ai-star></ai-star>
</td>
```

Để sử dụng nested component chúng ta vừa xây dựng, tại template của component cha (ở đây là product-list.component) chúng ta thay thế đoạn code template hiển thị số star rating thay bằng nested component **ai-star**.

product-list.component.html

```
<tbody>
  <tr *ngFor='let product of products | productFilter:listFilter'>
    <td>
      <img *ngIf='showImage'
        [src]='product.imageUrl'
        [title]='product.productName'
        [style.width.px]='imageWidth'
        [style.margin.px]='imageMargin'>
    </td>
```

```

<td>{{ product.productName }}</td>
<td>{{ product.productCode | lowercase }}</td>
<td>{{ product.releaseDate }}</td>
<td>{{ product.price | currency:'USD':true:'1.2-2' }}</td>
// thay thế <td>{{ product.starRating }}</td>
<td>
    <ai-star></ai-star>
</td>
</tr>
</tbody>

```

Dĩ nhiên chúng ta cũng không quên bao StartComponent cho AppModule biết rằng StarComponent đã sẵn sàng để sử dụng.

app.module.ts

```

@NgModule({
  imports: [
    BrowserModule,
    FormsModule
  ],
  declarations: [
    AppComponent,
    ProductListComponent,
    ProductFilterPipe,
    StarComponent
  ],
  bootstrap: [ AppComponent ]
})

```

Bây giờ hãy xem kết quả mà chúng ta đạt được cho đến lúc này.

Acme Product Management

Product List					
Filter by: <input type="text"/> Filtered by:					
Show Image	Product	Code	Available	Price	5 Star Rating
	Garden Cart	gdn-0023	March 18, 2016	\$32.99	★★★★★
	Hammer	tbx-0048	May 21, 2016	\$8.90	★★★★★

Bởi vì number start rating chúng ta đang để hardcoded. Do đó số sao biểu diễn trên view lúc này luôn là 5 sao. Chúng ta sẽ làm cho chúng hiển thị đúng như những gì mà chúng ta mong muốn ở bài học tiếp theo đây.

6.4 TRUYỀN DỮ LIỆU TỚI MỘT NESTED COMPONENT (@INPUT)

Nếu một nested component muốn nhận input từ component chứa nó, nó phải cung cấp thuộc tính để làm điều đó. Angular nói rằng, nested component chỉ ra một thuộc tính để nó có thể nhận input từ component chứa đó là "@Input" decorator. Trong ví dụ này, chúng ta muốn rating number truyền vào trong nested component, vì vậy chúng ta đánh dấu thuộc tính này với @Input decorator.

product-list.component.ts

```
@Component({
  selector: 'pm-products',
  templateUrl: 'product-list.component.html'
})
export class ProductListComponent { }
```

product-list.component.html

```
<td>
  <ai-star [rating]='product.starRating'>
  </ai-star>
</td>
```

star.component.ts

```
@Component({
  selector: 'ai-star',
  templateUrl: 'star.component.html'
})
export class StarComponent {
  @Input() rating: number;
  starWidth: number;
}
```

Chúng ta cần lưu ý: để truyền dữ liệu từ template cha (product-list.component.html) chúng ta sử dụng dấu ngoặc vuông và thiết lập dữ liệu nguồn từ dữ liệu của component cha mà chúng ta muốn truyền đến nested component.

rating cũng chính là tên thuộc tính mà chúng ta khai báo trong Start Component, @Input() rating

Trong ví dụ của chúng ta, chúng ta chỉ gắn một thuộc tính cho nested component với @Input decorator nhưng chúng ta không chỉ giới hạn một thuộc tính, chúng ta có thể chỉ định nhiều thuộc tính input khi cần thiết.

star.component.ts

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

@Component({
  selector: 'ai-star',
  templateUrl: 'app/shared/star.component.html',
```

```

        styleUrls: ['app/shared/star.component.css']
    })
export class StarComponent implements OnChanges {
    @Input() rating: 4;
    starWidth: number;

    ngOnChanges(): void {
        this.starWidth = this.rating * 86 / 5;
    }
}

```

Bất cứ khi nào dữ liệu trên component chứa Start Component thay đổi, sự kiện lifecycle OnChanges sẽ được tạo ra và chiều rộng của sao sẽ được tính lại, số sao thích hợp sẽ được hiển thị lại. Nhưng nếu chúng ta muốn gửi dữ liệu từ nested component trở lại component cha ? Hãy tiếp tục theo dõi ở bài tiếp theo.

6.5 TRUYỀN DỮ LIỆU TỪ MỘT COMPONENT (@OUTPUT)

Chúng ta vừa thấy cách mà component cha truyền dữ liệu tới một nested component bằng cách ràng buộc tới thuộc tính của nested component thông qua @Input decorator. Nếu nested component muốn gửi thông tin trở lại component chứa nó, nó có thể phát ra một sự kiện. Chúng ta có thể sử dụng @Output decorator để đặt bất kỳ thuộc tính nào của nested component khiến chúng có thể truyền một event tới component cha. Dĩ nhiên thuộc tính này phải là một sự kiện. Dữ liệu chúng ta muốn truyền sẽ được kèm event này.

product-list.component.ts

```

@Component({
    selector: 'pm-products',
    templateUrl: 'product-list.component.html'
})
export class ProductListComponent {
    onNotify(message: string): void { }
}

```

product-list.component.html

```

<td>
    <ai-star [rating]='product.starRating'
              (notify)='onNotify($event)'>
    </ai-star>
</td>

```

star.component.ts

```

@Component({
    selector: 'ai-star',
    templateUrl: 'star.component.html'
})
export class StarComponent {
    @Input() rating: number;
    starWidth: number;
    @Output() notify: EventEmitter<string> =
        new EventEmitter<string>();

    onClick() {
        this.notify.emit('clicked!');
    }
}

```

star.component.html

```

<div (click)='onClick()'>
    ... stars ...
</div>

```

Trong Angular một event thì được định nghĩa với một đối tượng EventEmitter. Ở đây chúng ta sẽ tạo ra một thể hiện mới của EventEmitter. Hãy chú ý cú pháp ở start.component.ts , Chúng ta vừa khai báo cho Star Component một sự kiện, sự kiện này khi phát ra sẽ truyền cùng với data là kiểu string.

Hàm onClick sẽ phát ra sự kiện notify và gắn vào sự kiện này một dữ liệu kiểu chuỗi có giá trị "clicked".

Để lắng nghe sự kiện này trên component cha, chúng ta sử dụng dấu ngoặc đơn (notify) và gán vào cho nó một hàm để lắng nghe sự kiện này.

6.6 TỔNG KẾT

Checklist: Nested Component



Input decorator

- Attached to a property of any type
- Prefix with @; Suffix with ()

Output decorator

- Attached to a property declared as an EventEmitter
- Use the generic argument to define the event payload type
- Use the new keyword to create an instance of the EventEmitter
- Prefix with @; Suffix with ()

Checklist: Container Component



Use the directive

- Directive name -> nested component's selector

Use property binding to pass data to the nested component

Use event binding to respond to events from the nested component

- Use \$event to access the event payload passed from the nested component

7. SERVICE VÀ DEPENDENCY INJECTION

7.1 GIỚI THIỆU

Những component có vẻ như đã ổn, nhưng chúng ta làm gì với những dữ liệu hoặc logic cái mà không liên quan gì tới một khung nhìn cụ thể nào đó hoặc rằng chúng ta muốn chia sẻ một số dữ liệu nào đó qua các component? Chúng ta xây dựng những service. Trong chương này, chúng ta tạo một service và sử dụng dependency injection để inject service đó vào bất kỳ component nào cần nó. Những ứng dụng thường yêu cầu những service, cũng giống như một product data service hay một logging service. Những components của chúng ta phụ thuộc vào những services này để làm những công việc nặng.

Nhưng chính xác thì những services là gì ? Nó là một class với một mục đích cụ thể, và được sử dụng cho các tính năng sau:

- Thực hiện những chức năng độc lập với bất kỳ component cụ thể nào
- Cung cấp những dữ liệu chia sẻ hoặc logic giữa các component.
- Đóng gói các tương tác bên ngoài.

Bằng cách chuyển những trách nhiệm này từ component sang service, code sẽ trở nên dễ dàng để test và sử dụng lại. Trong chương này, chúng ta sẽ bắt đầu với tổng quan về các service và dependency injection làm việc trong Angular. Tiếp đến chúng ta sẽ xây dựng, đăng ký, và cách sử dụng service.

7.2 NÓ LÀM VIỆC NHƯ THẾ NÀO?

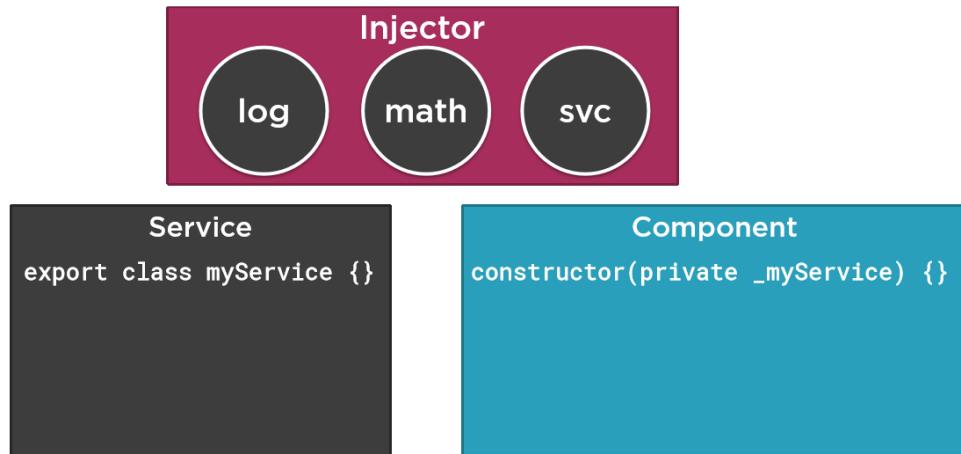
Trước khi chúng ta nhảy vào xây dựng một service, chúng ta hãy xem cách các services và dependency injection làm việc trong Angular. Trong sơ đồ dưới đây, bên trái là service của chúng ta, bên phải là component cần service này.



Có hai cách để component có thể làm việc với service này.

Component có thể tạo một thể hiện của lớp service và sử dụng chúng. Điều đó đơn giản, và nó sẽ hoạt động. Nhưng thể hiện này là địa phương bên trong component, do đó chúng ta không thể chia sẻ dữ liệu hoặc tài nguyên khác, và nó sẽ khó khăn để mô phỏng service cho testing. Ngoài ra, chúng ta có thể đăng ký service này với Angular. Angular sau đó tạo ra một thực thể duy nhất của service class

này, được gọi là singleton, và giữ nó. Cụ thể, Angular cung cấp một injector dựng sẵn. Chúng ta đăng ký những services với Angular injector, cái mà cung cấp một container những thể hiện service đã được tạo ra. Injector tạo ra và quản lý thể hiện duy nhất hoặc singleton của mỗi dịch vụ đã đăng ký theo yêu cầu.



Trong ví dụ này, Angular injector đang quản lý 3 thể hiện của 3 service khác nhau log, math và myService (viết tắt svc). Nếu component của chúng ta cần một service, component class định nghĩa một service như một dependency. Angular injector sau đó cung cấp hoặc inject thể hiện service class khi component class được khởi tạo. Tiến trình này được gọi là dependency injection. Khi đó Angular quản lý các thể hiện duy nhất, bất kỳ dữ liệu hoặc logic trong thể hiện đó được chia sẻ với tất cả các class sử dụng nó. Kỹ thuật này là cách được khuyến cáo sử dụng service, vì nó cung cấp quản lý tốt hơn cho các thể hiện của service. Nó cho phép chia sẻ dữ liệu và các tài nguyên khác, và nó dễ dàng hơn để mô phỏng các service cho các mục đích kiểm thử. Nay giờ chúng ta hãy xem xét một định nghĩa chính thức hơn về dependency injection.

Dependency Injection

A coding pattern in which a class receives the instances of objects it needs (called **dependencies**) from an external source rather than creating them itself.

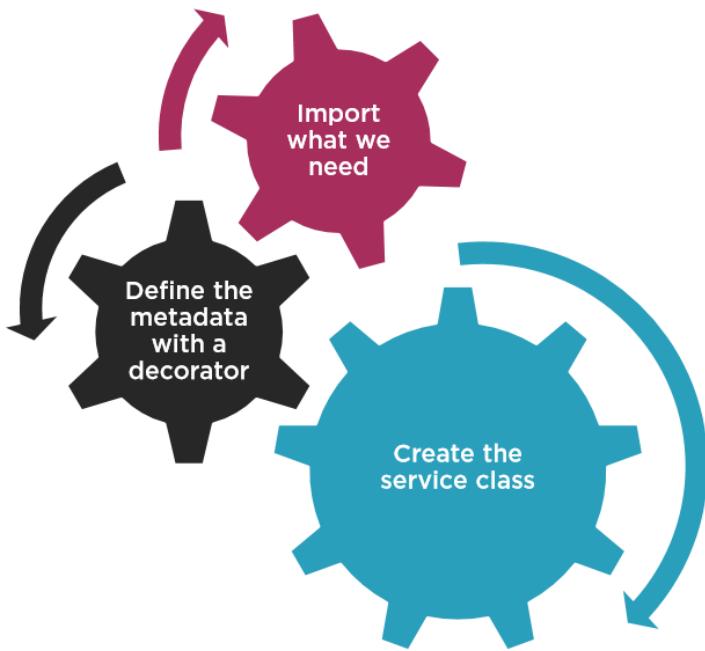
Bây giờ chúng ta đã hiểu ý tưởng về cách service và dependency injection làm việc trong Angular, hãy bắt tay vào xây dựng một service.

7.3 XÂY DỰNG MỘT SERVICE

Chúng ta đã sẵn sàng để xây dựng một service? Dưới đây là các bước.

1. Tạo một service class.
2. Định nghĩa metadata với một decorator.

3. Import những gì chúng ta cần.



Đây là những bước cơ bản tương tự với cách mà chúng ta đã theo để xây dựng các component và custom pipe. Hãy xem đoạn code của một service đơn giản dưới đây.

product.service.ts

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

@Injectable()
export class ProductService {

  getProducts(): IProduct[] {
  }

}
```

Đây là một class, chúng ta export nó vì vậy service này có thể được sử dụng ở bất cứ đâu trong ứng dụng. Class này chỉ có một phương thức, getProducts. Phương thức này sẽ trả về một danh sách của những sản phẩm (IProduct[]). Tiếp đến, chúng ta thêm một decorator cho service metadata. Khi xây dựng những service, chúng ta thường sử dụng Injectable decorator. Decorator này chỉ thực sự yêu cầu ở đây nên bản thân service này có một injected dependency. Tuy nhiên, mọi service class đều sử dụng Injectable decorator để làm rõ và nhất quán. Cuối cùng chúng ta import những gì chúng ta cần. Như

vậy là chúng ta đã hoàn thành việc xây dựng một service đơn giản. Hãy mở code ứng dụng mẫu để xem thêm.

7.4 ĐĂNG KÝ SERVICE

Như hình minh họa, chúng ta đăng ký service một Angular injector, và injector này cung cấp thể hiện của service tới bất kỳ class nào mà định nghĩa nó như một dependency. Vậy những gì thực sự liên quan đến việc đăng ký một service với injector? Để đăng ký một service chúng ta phải đăng ký một provider. Một provider là code mà có thể tạo hoặc trả về một service, thường là service class. Làm thế nào để đăng ký một provider? Chúng ta định nghĩa nó như một phần của component hoặc Angular module metadata.

Nếu chúng ta đăng ký một provider trong component metadata, Angular injector có thể inject service này vào component và bất kỳ những component con của nó. Vì vậy chúng ta nên chú ý để đăng ký provider ở mức thích hợp của cây thành phần ứng dụng.

Nếu chúng ta đăng ký một provider trong một Angular module, service được đăng ký với Angular injector tại gốc của ứng dụng, làm cho service sẵn sàng ở mọi nơi trong ứng dụng.

Chúng ta sẽ nói nhiều hơn về việc đăng ký một provider trong một Angular module khi chúng ta bao quát Angular module trong chi tiết những chương sau. Ngay bây giờ chúng ta sẽ chỉ bàn đến việc đăng ký một service với một component.

app.component.ts

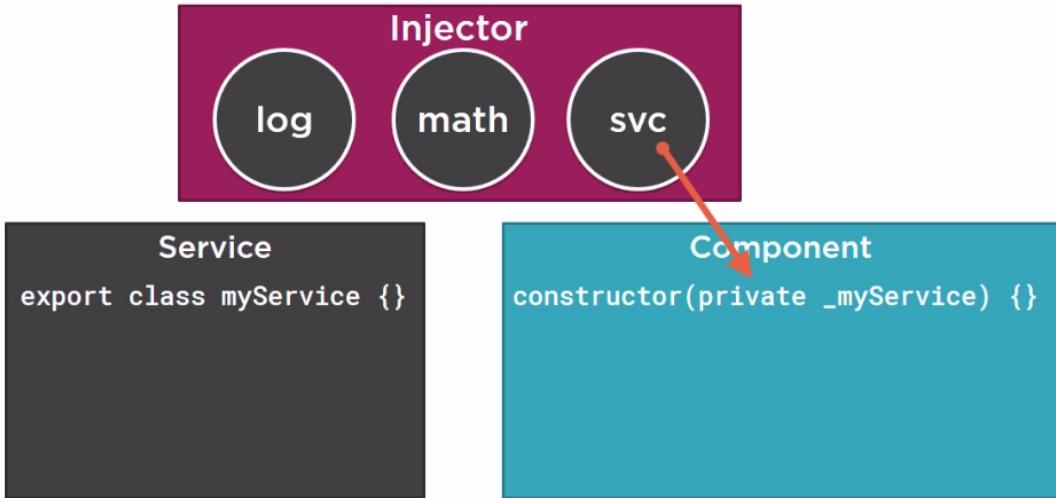
```
...
import { ProductService } from './products/product.service';

@Component({
  selector: 'pm-app',
  template: `
    <div><h1>{{pageTitle}}</h1>
      <pm-products></pm-products>
    </div>
  `,
  providers: [ProductService]
})
export class AppComponent { }
```

7.5 INJECTING THE SERVICE

Một lần nữa chúng ta xem lại minh họa dưới đây, chúng ta đã thấy làm thế nào để đăng ký một service với Angular injector. Bây giờ, chúng ta cần định nghĩa nó như một dependency vì vậy injector sẽ cung cấp thể hiện trong những class mà cần nó.

Injecting the Service



Vậy, làm thế nào để chúng ta làm dependency injection trong Angular? Vâng, câu hỏi tốt hơn là làm thế nào để chúng ta làm dependency injection trong Typescript?

Câu trả lời là, trong hàm khởi tạo (constructor). Mỗi một class có một constructor cái mà được thực thi khi một thể hiện của class được tạo ra. Nếu không có constructor rõ ràng nào được định nghĩa cho class, một constructor ẩn sẽ được sử dụng. Cho đến nay chúng ta vẫn chưa cần một constructor cụ thể, nhưng nếu chúng ta muốn chèn thêm các phụ thuộc (inject dependencies) như một thể hiện của một dịch vụ, chúng ta cần một constructor rõ ràng. Trong TypeScript, một constructor được định nghĩa bởi hàm constructor. Chúng ta sẽ đặt gì bên trong hàm constructor? Ít nhất có thể. Vì hàm constructor được thực hiện khi component được tạo ra. Nó chủ yếu được sử dụng để khởi tạo, và không nên cho những đoạn code chiếm nhiều thời gian thực hiện vào. Chúng ta xác định các phụ thuộc (dependencies) bằng cách xác định chúng như các tham số cho hàm constructor, như dưới đây.

product-list.component.ts

```
...
import { ProductService } from './products/product.service';

@Component({
  selector: 'pm-products',
  templateUrl: 'product-list.component.html'
})
export class ProductListComponent {
  private _productService;
  constructor(productService: ProductService) {
    _productService = productService;
  }
}
```

Ngoài ra chúng ta có thể khai báo ngắn gọn hơn cho hàm contructor như dưới đây.

product-list.component.ts

```
...
import { ProductService } from './products/product.service';

@Component({
  selector: 'pm-products',
  templateUrl: 'product-list.component.html'
})
export class ProductListComponent {

  constructor(private _productService: ProductService) {
  }
}
```

Như vậy chúng ta đã tạo được một thể hiện của ProductService là _productService. Và chúng ta đã có thể sử dụng thể hiện của service này. Tham khảo code ứng dụng mẫu để biết thêm về cách áp dụng service vào thực tế ứng dụng của chúng ta.

7.6 TỔNG KẾT

Checklist: Creating a Service



Service class

- Clear name
- Use PascalCasing
- Append "Service" to the name
- export keyword

Service decorator

- Use Injectable
- Prefix with @; Suffix with ()

Import what we need

Checklist: Registering a Service in a Component



Select the appropriate level in the hierarchy

- Root component if service is used throughout the application
- Specific component if only that component uses the service
- Otherwise, common ancestor

Component metadata

- Set the providers property
- Pass in an array

Import what we need

Checklist: Dependency Injection

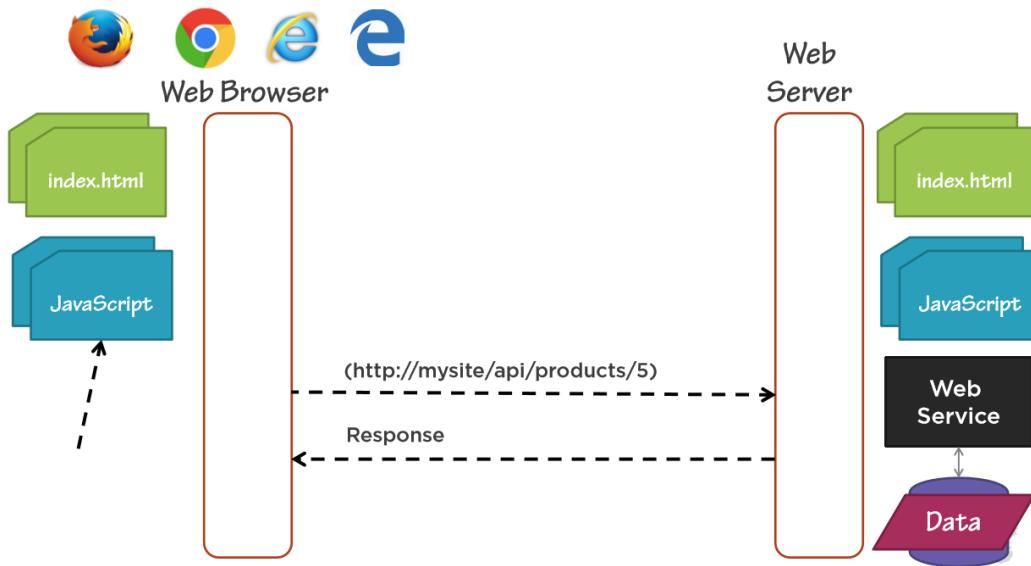


- Specify the service as a dependency
- Use a constructor parameter
- Service is injected when component is instantiated

8. TRUY XUẤT DỮ LIỆU BẰNG HTTP

8.1 GIỚI THIỆU

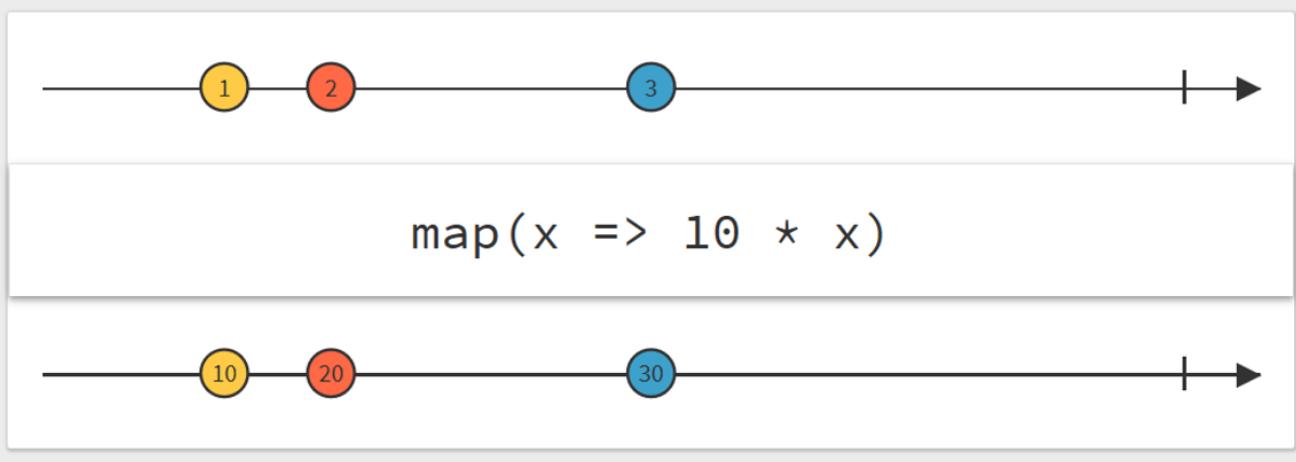
Trong chương này, chúng ta sẽ học cách sử dụng Http với observables để lấy dữ liệu. Hầu hết các ứng dụng Angular có dữ liệu sử dụng Http. Các ứng dụng gửi một request http get tới một dịch vụ web. Dịch vụ web truy xuất dữ liệu, thường sử dụng một cơ sở dữ liệu và trả về cho ứng dụng một response có chứa dữ liệu. Các ứng dụng sau đó sẽ xử lý dữ liệu này.



8.2 OBSERVABLES VÀ REACTIVE EXTENSIONS

Các observable giúp chúng ta quản lý dữ liệu không đồng bộ, chẳng hạn như dữ liệu đến từ một back-end service. Observable đối xử với các event như một collection. Chúng ta có thể nghĩ về một observable như một mảng mà các mục đến không đồng bộ theo thời gian. Observable là tính năng được đề xuất cho ES2016, một phiên bản tiếp theo của Javascript. Để sử dụng các observable lúc này, Angular sử dụng một thư viện bên thứ ba gọi là reactive extensions. Đừng nhầm lẫn điều này với React, đó là một điều hoàn toàn khác. Các observable được sử dụng trong chính Angular, bao gồm hệ thống sự kiện của Angular và Http clients service, đó là lý do mà chúng ta đề cập đến chúng ở đây. Các observable cho phép chúng ta thao tác trên tập hợp các sự kiện với các operator. Operator là các phương thức trên các observables để tạo nên những observable mới. Mỗi operator biến đổi các nguồn observable theo một cách nào đó. Các operator không chờ đợi tất cả các giá trị trả về và xử lý cùng một lúc. Thay vào đó các operator trên các observable xử lý mỗi giá trị chừng nào nó được phát ra. Một số ví dụ của operator bao gồm map, filter, take, and merge. Các chuỗi dữ liệu có thể có nhiều hình thức, chẳng hạn như response từ back-end web service, một tập hợp của hệ thống thông báo hoặc hệ thống các sự kiện chẳng hạn như input của người dùng. Trong minh họa dưới đây, thể hiện operator map biến đổi chuỗi dữ liệu 1,2,3 thành 10,20,30.

Interactive diagrams of Rx Observables



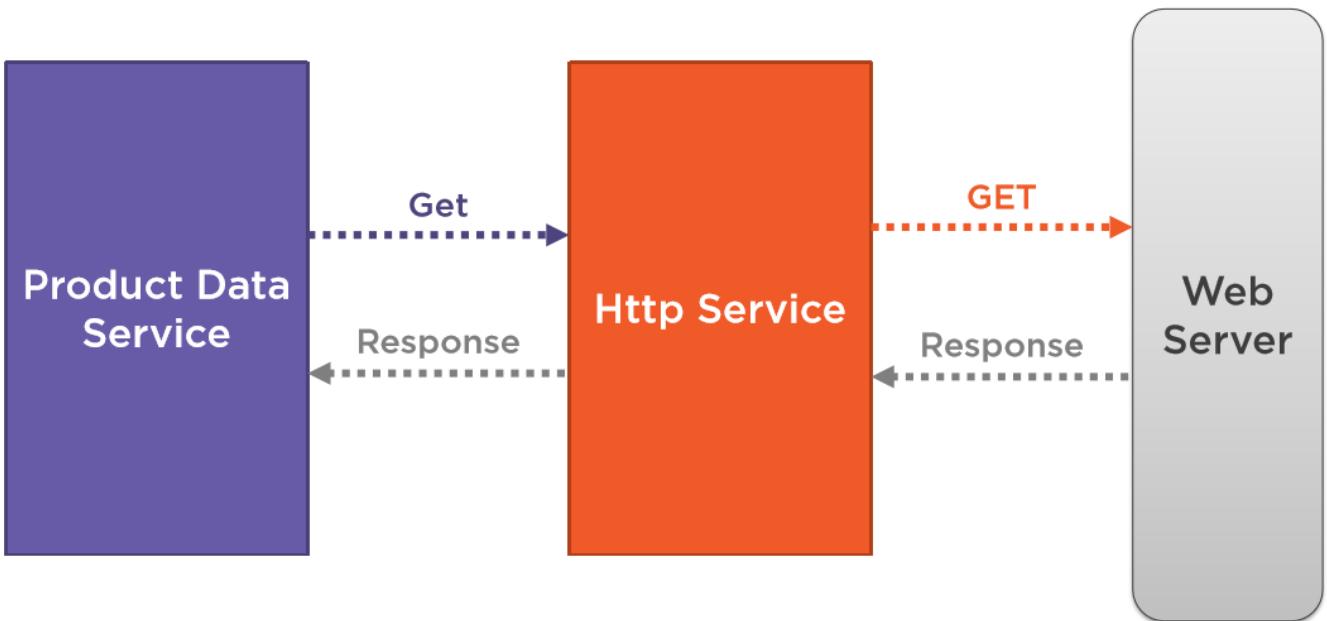
Bạn có thể đã làm việc với dữ liệu không đồng bộ trong Javascript trước đây bằng cách sử dụng `Promise`. `Observable` thì khác với `Promise` trong một số cách.

Promise	Observable
<ul style="list-style-type: none">- Trả về một giá trị duy nhất trong tương lai- Không lazy- Không thể hủy	<ul style="list-style-type: none">- Phát ra nhiều giá trị không đồng bộ theo thời gian- Lazy- Có thể hủy đăng ký- Hỗ trợ <code>map</code>, <code>filter</code>, <code>reduce</code>, v.v..

Tuy nhiên bạn vẫn có thể sử dụng `Promise` thay vì `Observable` khi gọi `Http` trong Angular nếu bạn muốn, nhưng trong chương này chúng ta sẽ làm việc với `Http` thông qua các `Observable`.

8.3 GỬI MỘT YÊU CẦU HTTP

Chúng ta thường đóng gói dữ liệu truy cập của ứng dụng vào một service có thể được sử dụng bởi bất kỳ component nào hoặc service khác cần nó. Trong chương trước, chúng ta đã làm điều đó, nhưng product service vẫn chứa một danh sách sản phẩm với hard-coded. Thay vào đó chúng ta muốn gửi một yêu cầu `Http` tới một back-end web server để lấy danh sách sản phẩm này.



Angular là cung cấp một Http service cái mà cho phép chúng ta có thể giao tiếp với một back-end web server. Sử dụng nó tương tự như Http request protocol. Hình minh họa trên sẽ giúp chúng ta hình dung về cách mà chúng ta sẽ gửi một Http request. Cụ thể hơn chúng ta sẽ thảo luận về đoạn code sử dụng http service để gọi một http request.

product.service.ts

```

...
import { Http } from '@angular/http';

@Injectable()
export class ProductService {
  private _productUrl = 'www.myWebService.com/api/products';

  constructor(private _http: Http) { }

  getProducts() {
    return this._http.get(this._productUrl);
  }
}

```

Ở đây, `_productUrl` chính là địa chỉ web server mà chúng ta sẽ gọi đến. bên trong hàm `constructor` chúng ta tạo ra một thể hiện của `Http service`, như cách mà chúng ta inject một service như ở chương trước đó. Cuối cùng để lấy được danh sách sản phẩm từ web server, chúng ta viết một hàm `getProducts` trong hàm này sẽ trả về một observable, kết quả nhận được từ thể hiện của `Http service` gọi hàm `get` đến địa chỉ web server.

Cũng cần lưu ý, vì ProductService class được định nghĩa như một service, nên chúng ta cũng không quên đặt trước nó một decorator là @Injectable(). Và chúng ta cũng phải import Http để có thể sử dụng service này cả trong component và AppModule.

```
app.module.ts
...
import { HttpClientModule } from '@angular/http';

@NgModule({
  imports: [
    BrowserModule,
    FormsModule,
    HttpClientModule ],
  declarations: [
    AppComponent,
    ProductListComponent,
    ProductFilterPipe,
    StarComponent ],
  bootstrap: [ AppComponent ]
})
export class AppModule { }
```

8.4 XỬ LÝ NGOẠI LỆ

Như bạn có thể tưởng tượng, có rất nhiều thứ có thể dẫn đến lỗi khi giao tiếp với một backend service. Mọi thứ từ việc gửi đi một yêu cầu không hợp lệ cho đến việc bị mất kết nối với web service. Vì vậy chúng ta phải xử lý một vài ngoại lệ trong những trường hợp này.

```
product.service.ts
...
import 'rxjs/add/operator/do';
import 'rxjs/add/operator/catch';
...

getProducts(): Observable<IProduct[]> {
  return this._http.get(this._productUrl)
    .map((response: Response) => <IProduct[]>response.json())
    .do(data => console.log('All: ' + JSON.stringify(data)))
    .catch(this.handleError);
}

private handleError(error: Response) { }
```

Chúng ta thêm hàm catch như hình minh họa và truyền vào nó một phương thức xử lý lỗi. phương thức xử lý lỗi sẽ có một tham số truyền vào, một error response object. Trong phương thức đó, chúng

ta sẽ xử lý lỗi sao cho thích hợp, như việc chúng ta gửi thông báo lỗi này cho một log service để hiển thị lỗi này ra ngoài. Ngoài ra, dễ dàng nhận thấy hàm do trong ví dụ trên có thể lấy data và "xem trộm" nó.

Chú ý, để có thể sử dụng 2 operator do và catch chúng ta cần phải import 2 thư viện là rxjs/add/operator/do và rxjs/add/operator/catch.

Như vậy là chúng ta đã có thể xử lý những ngoại lệ theo những gì chúng ta cần. Ngay bây giờ hãy thử làm một ví dụ và so sánh với code ứng dụng mẫu của chúng ta.

8.5 ĐĂNG KÝ MỘT OBSERVABLE

Nếu như đã quen thuộc với Promise, bạn sẽ biết rằng chúng ta sẽ nhận được một kết quả từ promise, bằng cách gọi hàm then. Hàm then có 2 tham số một giá trị là function cái mà sẽ gọi khi promise hoàn thành thành công và một error function khi promise đó gặp bất kỳ lỗi nào. Như ví dụ minh họa dưới đây:

```
x.then(valueFn, errorFn) //Promise  
x.subscribe(valueFn, errorFn) //Observable  
x.subscribe(valueFn, errorFn, completeFn) //Observable  
let sub = x.subscribe(valueFn, errorFn, completeFn)
```

Nếu chúng ta định nghĩa x là một observable thay cho promise, chúng ta đơn giản chỉ cần thay đổi then thành subscribe. Vì observable xử lý nhiều giá trị theo thời gian, nên hàm valueFn được gọi mỗi khi observable emit. Trong một số trường hợp, chúng ta muốn biết khi nào observable chạy xong, chúng ta thêm completeFn vào tham số thứ 3 của hàm subscribe. Hàm này sẽ được gọi khi observable chạy xong. Hàm subscribe có giá trị trả về là một subscription, chúng ta có thể sử dụng subscription để gọi unsubscribe và huỷ bỏ subscription nếu cần.

Bây giờ hãy quay lại với ứng dụng demo của chúng ta, chúng ta đã có một productService ở bài trước. Và lúc này chúng ta sẽ sử dụng nó để lấy những sản phẩm từ back-end service.

product-list.component.ts

```
ngOnInit(): void {  
    this._productService.getProducts()  
        .subscribe(products => this.products = products,  
                 error => this.errorMessage = <any>error);  
}
```

Như đã biết, hàm getProducts mà chúng ta viết trả về cho chúng ta một observable. Và một observable sẽ không chạy cho đến khi chúng ta subscribe chúng. Như đoạn code trên, danh sách sản

phẩm sẽ được gán khi mà back-end service trả về giá trị cho productService, lúc này observable sẽ emit giá trị nhận được cho nơi mà nó subscribe.

8.6 TỔNG KẾT

Http Checklist: Service



Import what we need

Define a dependency for the http client service

- Use a constructor parameter

Create a method for each http request

Call the desired http method, such as get

- Pass in the Url

Map the Http response to a JSON object

Add error handling

Http Checklist: Subscribing



Call the subscribe method of the returned observable

Provide a function to handle an emitted item

- Normally assigns a property to the returned JSON object

Provide an error function to handle any returned errors

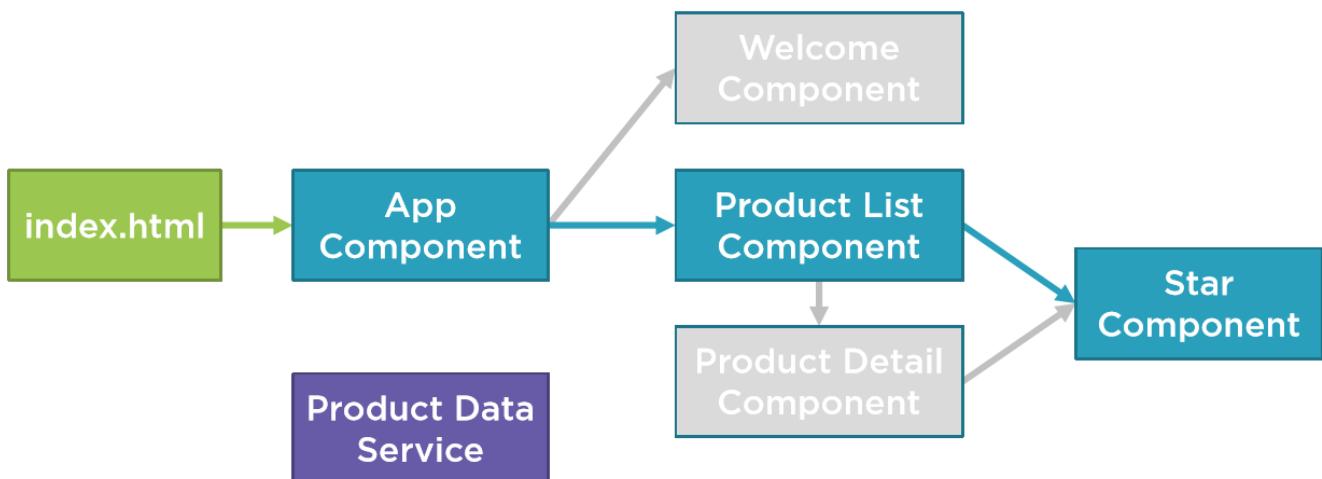
9. KHÁI NIỆM CƠ BẢN VỀ ĐỊNH HƯỚNG (NAVIGATION) VÀ ĐỊNH TUYẾN (ROUTING)

9.1 GIỚI THIỆU

Chỉ mới một view thì không thể làm thành một ứng dụng. Trong hai chương tới đây chúng ta sẽ định nghĩa các routers để điều hướng giữa nhiều view trong ứng dụng của chúng ta. Thông thường khi người dùng tiếp cận với ứng dụng của chúng ta, người dùng sẽ cần thao tác trên nhiều view khác nhau, mỗi view sẽ có những bộ dữ liệu và những bối cảnh kèm theo nó. Routing trong Angular cung cấp cho người dùng một cách để điều hướng giữa nhiều view trong ứng dụng, và số lượng view thì không giới hạn.

Với chương này, chúng ta sẽ bắt đầu với một cái nhìn tổng quan về cách mà routing làm việc trong Angular. Chúng ta sẽ kiểm tra xem làm thế nào để cấu hình các routes, các action của router và nơi chúng ta sẽ đặt các router của các component.

Hiện tại, ứng dụng của chúng ta nhúng ProductListComponent như một Component lồng trong App Component và mặc định người dùng sẽ đến Product List view. Chúng ta sẽ muốn xác định các tuyến đường khác để người dùng có thể điều hướng đến welcome view, product list view và product detail view. Chúng ta đã có code của Welcome Component và Product List Component, do đó trong chương này chúng ta sẽ xây dựng Product Detail Component, kết thúc với chương này, chúng ta sẽ có được một ứng dụng đơn giản với các routes dẫn đến nhiều view khác nhau. Hãy xem lại một chút kiến trúc của ứng dụng chúng ta.

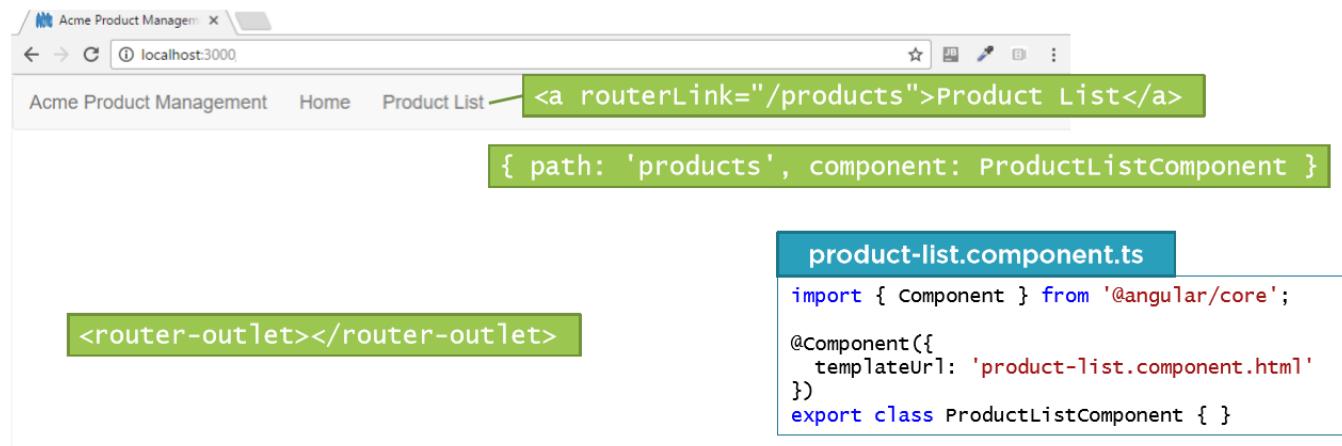


9.2 CÁCH ROUTING HOẠT ĐỘNG

Một ứng dụng Angular là một “single page application”. Điều đó có nghĩa là tất cả view của chúng ta sẽ hiển thị trong cùng một trang. Thường được định nghĩa trong tệp index.html. Vì vậy, cho dù ứng dụng có 5 view, 10 view hay 100 view thì cũng lần lượt xuất hiện trên một trang index đó. Tuy nhiên làm thế nào để quản lý được khi nào thì view nào xuất hiện. Đó chính là mục đích của routing.

Chúng ta cấu hình một route cho mỗi một component, cái mà chúng ta muốn hiển thị lên trang. Như một phần của ứng dụng chúng ta đang thiết kế, chúng ta cung cấp một menu, một toolbar, những button, images, hay data link cho phép người dùng chọn một view để hiển thị. Chúng ta buộc một route với mỗi một lựa chọn hoặc một hành động, khi mà người dùng chọn hay thực hiện một hành động lên chúng, thì route đi kèm sẽ được kích hoạt. Route được kích hoạt sẽ hiển thị giao diện của component mà nó được cấu hình.

Để hình dung rõ hơn, chúng ta sẽ nhìn qua quá trình đó với một minh họa dưới đây.



Dưới thanh nhập địa chỉ của browser là menu chúng ta thêm vào ứng dụng mẫu. chúng ta buộc một route vào mỗi menu option sử dụng một router directive có sẵn là **routerLink**.

Ví dụ, khi người dùng click lên product list thì Angular router sẽ điều hướng tới route của product. Url của trình duyệt cũng thay đổi để phù hợp với router này là sẽ trở thành <http://localhost:3000/products>. Khi đó Angular route tìm kiếm một route được định nghĩa giống với path segment ('products') lúc này ProductListComponent sẽ được tải lên trong trường hợp này có nghĩa là templateUrl trong product-list.component.ts sẽ được hiển thị lên page.

Có một thắc mắc là chúng ta sẽ không biết ProductListComponent sẽ được load lên đâu? Angular có định nghĩa một routing directive dựng sẵn là <router-outlet></router-outlet> . Đây chính là nơi mà ProductListComponent sẽ xuất hiện.

Và đó là cách mà routing làm việc. Chúng ta sẽ đi sâu vào chi tiết làm rõ các bước này và thử chúng cho các component khác của ứng dụng mẫu. Nhưng trước hết, chúng ta hãy hoàn thành component còn thiếu là ProductDetailComponent, nó sẽ trông giống như dưới đây.

Product Detail: Video Game Controller

Name:	Video Game Controller
Code:	GMG-0042
Description:	Standard two-button video game controller
Availability:	October 15, 2015
Price:	\$35.95
5 Star Rating:	



[Back](#)

9.3 CẤU HÌNH ROUTES

Routing dựa trên component, vì vậy chúng ta xác định những thành phần mà chúng ta muốn routing tới chúng và định nghĩa từng component một. Hãy cùng xem cách điều này được thực hiện.

Một ứng dụng Angular có một router mà được quản lý bởi Angular router service và chúng ta đã biết rằng để sử dụng service chúng ta cần đăng ký module cung cấp service này, cũng tương tự như cách chúng ta đăng ký HttpModule. Angular cung cấp cho chúng ta RouterModule, để khai báo chúng ta sẽ xem đoạn code mẫu dưới đây.

```
//app.module.ts
import { RouterModule } from '@angular/router';
@NgModule({
  imports: [
    BrowserModule,
    FormsModule,
    HttpModule,
    RouterModule.forRoot([]) // cấu hình trong mảng []
    //RouterModule.forRoot([], {useHash: true}) sử dụng # trong url
  ],
  declarations: [
    ...
  ],
  bootstrap: [ AppComponent ]
})
export class AppModule { }
```

Với đoạn đăng ký trên, chúng ta đã sẵn sàng để cấu hình những routes.

```
// Configuring Routes
[
```

```

{ path: 'products', component: ProductListComponent },
{ path: 'product/:id', component: ProductDetailComponent },
{ path: 'welcome', component: WelcomeComponent },
{ path: '', redirectTo: 'welcome', pathMatch: 'full' },
{ path: '**', component: PageNotFoundComponent }
]

```

Router phải được cấu hình với một danh sách các route được định nghĩa. Mỗi định nghĩa xác định một đối tượng route. Mỗi route yêu cầu một path(dường dẫn). Thuộc tính path này định nghĩa một phần của URL cho route. Khi route được kích hoạt, path này sẽ nối vào URL ứng dụng của chúng ta. Người dùng có thể gõ thêm vào hoặc bookmark Url này để có thể trở lại trực tiếp vào view của thành phần liên quan tới path này.

Trong hầu hết các trường hợp, mỗi một path chúng ta sẽ chỉ định với một component, là một component mà liên quan đến route. Template của component này sẽ được hiển thị khi route được kích hoạt. Hãy xem qua tất cả các đối tượng route được định nghĩa trong list danh sách ví dụ trên.

- Route đầu tiên chỉ đơn giản là định tuyến một path URL cụ thể tới ProductListComponent.
`{ path: 'products', component: ProductListComponent }`
- :id trong Router thứ hai đại diện cho một tham số của route. Trang chi tiết sản phẩm sẽ hiển thị chi tiết của một sản phẩm, do đó id cần để biết sản phẩm nào được hiển thị .
`{ path: 'product/:id', component: ProductDetailComponent }`
- Route thứ ba đơn giản là để định tuyến cho WelcomeComponent.
`{ path: 'welcome', component: WelcomeComponent }`
- Route thứ tư định nghĩa một định tuyến mặc định khi chúng ta không thêm bất cứ một path nào vào Url gốc. Trong trường hợp này khi người dùng nhập vào một Url gốc ứng dụng sẽ hiển thị WelcomeComponent
`{ path: '', redirectTo: 'welcome', pathMatch: 'full' }`
- Trong route cuối cùng, chúng ta thấy dấu 2 sao '**', điều này có ý nghĩa bất kỳ path nào trên Url không giống với những route đã cấu hình thì sẽ active route này. Và khi đó PageNotFoundComponent sẽ hiển thị.
`{ path: '**', component: PageNotFoundComponent }`

Ngay bây giờ, chúng ta đã có thể cấu hình 3 component Welcome, ProductList, và ProductDetail theo như mục đích mà chúng ta đặt ra ở chương này. Hãy bắt đầu ngay trên source code ứng dụng mẫu của chúng ta.

9.4 KẾT NỐI CÁC ROUTE TỚI CÁC ACTION.

Với routing, người dùng có thể điều hướng thông qua ứng dụng bằng nhiều cách.

- Người dùng có thể click vào một menu, link, hình ảnh, button
- Gõ url vào trong address bar
- Click vào button chuyển tiếp hoặc button quay lại của trình duyệt.

Cách thứ hai và thứ ba hoạt động dựa vào đặc tính kỹ thuật của browser, do đó chúng ta sẽ chỉ tìm hiểu làm thế nào chúng ta thực hiện được cách thứ nhất.

Ứng dụng của chúng ta có rất nhiều menu-option và sub-options, nhưng chúng ta sẽ chỉ tập trung vào menu Home và ProductList trong khuôn khổ bài học này.

Để buộc một route với một menu option, chúng ta sử dụng routerLink như đoạn code dưới đây. routerLink là một attribute directive.

```
//app.component.ts
@Component({
  selector: 'pm-app',
  template: `
    <ul class='nav navbar-nav'>
      <li><a [routerLink]="/welcome">Home</a></li>
      <li><a [routerLink]="/products">Product List</a></li>
    </ul> `
})
```

Chỉ đơn giản như vậy là chúng ta đã có thể kết nối các route tới các action mà chúng ta mong muốn.

9.5 XẾP CHỖ CÁC VIEW

Khi các route được kích hoạt, component liên kết với route sẽ được hiển thị, nhưng nó sẽ hiển thị ở đâu? Làm thế nào để chúng ta xác định nơi chúng ta muốn component được định tuyến hiển thị?

Chúng ta sử dụng router outlet directive.

Chúng ta đặt directive này vào trong template component chính, Component được định tuyến sẽ được xuất hiện ở vị trí đặt directive này. Như ví dụ dưới đây.

```
//app.component.ts
@Component({
  selector: 'pm-app',
  template: `
    <ul class='nav navbar-nav'>
      <li><a [routerLink]="/welcome">Home</a></li>
```

```
<li><a [routerLink]=["/products"]>Product List</a></li>
</ul>
<router-outlet></router-outlet> `
```

9.6 TỔNG KẾT

Hiển thị các component



Nest-able components

- Định nghĩa một selector
- Lồng trong một component khác
- Không định tuyến

Routed components

- Không selector
- Cấu hình các định tuyến
- Buộc các định tuyến vào các hành động.

Làm một route



Cấu hình các route

Buộc các route với các action

Xếp chỗ view

Cấu hình Route



Định nghĩa thành phần gốc

Thêm RouterModule

- Thêm từng route
(RouterModule.forRoot)
- Sử dụng # hoặc không

path: Url segment cho route

- Không có dấu gạch chéo
- '' dành cho route mặc định
- '**' cho route đại diện

component

- Không phải tên dạng chuỗi, không bao gồm dấu ngoặc kép

Buộc các định tuyến với các hành động



Thêm RouterLink directive như một thuộc tính

- Thuộc tính có thể click chuột
- Đặt trong dấu ngoặc vuông
-

Ràng buộc tới một mảng tham số những liên kết

- Thành phần đầu tiên là một path (đường dẫn)
- Tất cả thành phần khác là tham số route

Xếp chỗ view



Thêm RouterOutlet directive

- Xác định nơi để hiển thị component được định tuyến.
- Xác định bên trong template component chủ