**2. TYPESCRIPT**  \_**The Last Shinobi ϟ\_**

- **TypeScript** – là 1 **superSet**, 1 phiên bản nâng cao của **JavaScript** ,là 1 dự án mã nguồn mở đc **Microsoft** phát triển, nhằm khắc phục các nhược điểm của **JS** và nâng cấp thêm các tính năng mới.

- **Kiểu dữ liệu trong TS:**

**+ Bao gồm các kiểu dữ liệu của JS:** string, number, boolean, object, symbol, undefined, null,…

🡪 Khai báo **biến**:





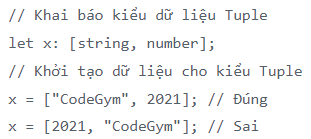


🡪 Khai báo **mảng**:

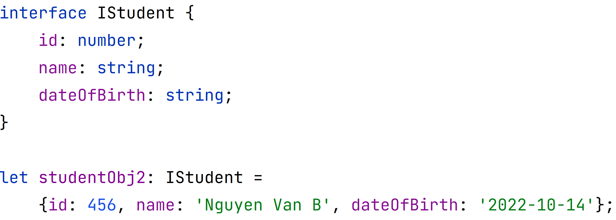


+ **Có thêm các kiểu dữ liệu mới:**

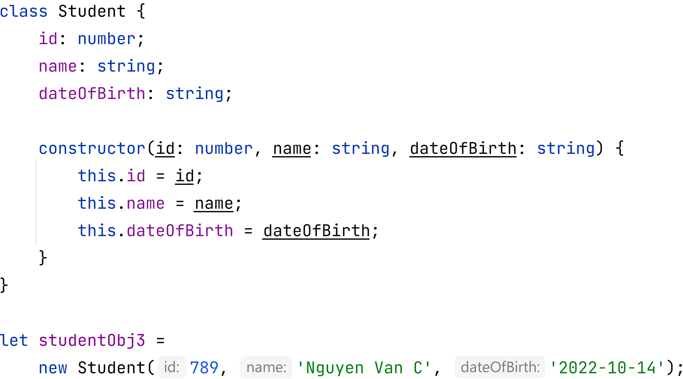
- **Tuple** – cho phép khai báo mảng với các giá trị có kiểu dữ liệu đã biết (giống như 1 mảng có độ dài cố định, và kiểu dữ liệu của các phần tử trong mảng là cố định).



- **Interface**



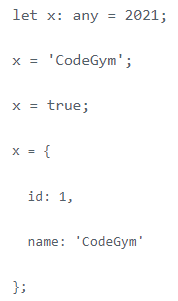
- **Class**



- **Enum** – 1 biến có thể là một tập hợp các hằng số được định sẵn.



- **Any** – có thể áp dụng cho bất kì kiểu dữ liệu nào, sử dụng khi không biết chắc kiểu dữ liệu (nên hạn chế dùng vì nó không tường minh).



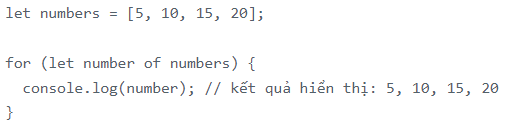
- **Cấu trúc điều kiện và vòng lặp** (cũng như **JS**):

+ Cấu trúc điều kiện: **if-else**, **switch-case**, toán tử ba ngôi.

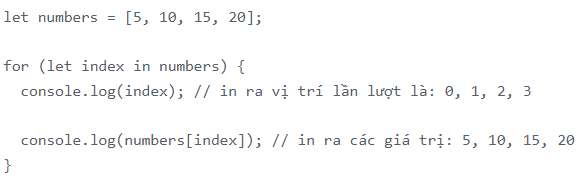
+ Cấu trúc lặp:

- **for** thường.

- **for of** – trả lại từng phần tử của tập hợp.



- **for in** – duyệt qua các phần tử của mảng và trả về vị trí **index** của các phần tử trong mảng.



- **while.**

- **do-while.**

**- Phân biệt sự khác nhau giữa var, let, const:**

- **Giống**: đều là từ khóa dùng để khái báo biến, tạo 1 vùng nhớ chứa dữ liệu.

- **Khác** (cái nào khai báo lại đc? cái nào ghi đè đc? cái nào có cơ chế **hoisting**? phạm vi truy cập?) 🡪 xem lại kiến thức **module 1** nha mấy fen ☺. Mấy fen bt rồi ☹ but để Minchou lo

|  |  |  |
| --- | --- | --- |
| **var** | **let** | **const** |
| - Có thể khai báo đè. (lấy giá trị sau)  - Có cơ chế **hoisting** cho phép sử dụng trc khi khởi tạo.  - Phạm vi truy cập của biến nằm trong **function scope**. | - K khai báo lại.  - K có cơ chế **hoisting**.  - Phạm vi truy cập của biến nằm trong **block scope.** | - Phải gán giá trị khi khởi tạo  - Không thể ghi đè (giống let)  - Phạm vi truy cập của biến nằm trong **block scope.** |

- **Phân biệt TS và JS:**

+ **Giống**: cả 2 ngôn ngữ đều tuân theo đặc tả **ECMAScript** nên cú pháp có sự tương đồng rất lớn, giống nhau về 1 số kiểu dữ liệu, các từ khóa khai báo biến (**var**, **let**, **const**),…

+ **Khác**:

|  |  |
| --- | --- |
| **TypeScript** | **JavaScript** |
| - Là 1 **superSet** của **JS**.  - **File TS** có đuôi là \***.ts**  - **Code TS** đc biên dịch thành **JS** thuần.  - Kiểu dữ liệu rõ ràng hơn.  - Kiểm tra kiểu của các biến khi **compile-time**.  - Dùng trong các dự án lớn.  - Bổ sung thêm các kiểu dữ liệu tĩnh, các lớp hướng đối tượng, các phiên bản **ECMAScript** mới nhất, hỗ trợ **Angular2** và **NodeJs**. | - Là ngôn ngữ kịch bản.  - **File JS** có đuôi là \***.js**  - Không cần biên dịch lại.  - Kiểu dữ liệu không rõ ràng.  - Kiểm tra kiểu của các biến lúc **run-time**.  - Thích hợp với các dự án nhỏ. |

- **Các lệnh Transpile TS – chuyển mã TS thành mã JS** (**transpile** là biên dịch trong cùng 1 ngôn ngữ họ hàng, khác với **compile** là biên dịch từ 1 ngôn ngữ này sang 1 ngôn ngữ khác)

🡪 Tạo **file** \***.ts**

🡪 Di chuyển **terminal / cmd** vào thư mục chứa **file** \***.ts**, gõ các lệnh sau:

🡪 **tsc** tên\_file

🡪 **node** tên\_file

- **Bất đồng bộ (Asychronous)** – tức là các dòng **code** sẽ không kết thúc theo trình tự, vì phải đợi sử lý ở **web service**.

(Không phải lúc nào cũng xử lý **bất đồng bộ**, vì thực tế có nhiều **web** có các phần nội dung độc lập với nhau, nội dung nào xuất hiện trước cũng k ảnh hưởng đến các phần khác).

- **Các cơ chế xử lý bất đồng bộ trong TS** – có 3 cách:

+ **Callback** (**ES5**) – gọi **function** trong **function**.

🡪 Dễ xảy ra hiện tượng **callback hell** (việc lạm dụng **callback** khiến **code** của chúng ta trông giống như một kim tự tháp vậy).

+ **Promise** (**ES6**) – xử lý kết quả (thành công / thất bại) của một hành động cụ thể.

🡪 Cách tạo:

let promise = new Promise(function (resolve, reject) {

});

Trong đó:

+ **resolve** – là một hàm **callback** xử lý cho hành động thành công.

+ **reject** – là một hàm **callback** xử lý cho hành động thất bại.

🡪 Khi một **Promise** được khởi tạo thì nó có 1 trong 3 trạng thái sau:

+ **Pending** – hành động đang chờ xử lý hoặc bị từ chối

+ **Fulfilled** – hành động xử lý xong và thành công.

+ **Rejected** – hành động xử lý xong và thất bại.

+ **Async / Await** (**ES7**) – bản chất cũng là **Promise**.

🡪 Trả về đoạn mã **clean** hơn, dễ đọc, dễ **maintain**.

🡪 Xử lý lỗi đơn giản hơn nhiều và nó sử dụng **try / catch** giống như trong bất kỳ đoạn mã đồng bộ thông thường.

🡪 Đơn giản hơn trong việc **debug**, tìm lỗi.

+ Ngoài ra, những bài sau sẽ đc học về **Observable**.

🡺 Từ những nội dung trên, ta suy ra đc **“tại sao nên sử dụng TS, ưu điểm của TS?”**:

+ **TS** là phiên bản nâng cao của **JS**, nó nâng cấp, khắc phục những nhược điểm của **JS** (như đã so sánh ở trên).

+ Hỗ trợ lập trình hướng đối tượng (**OOP**).

+ Dễ dàng, tối ưu trong phát triển các dự án lớn, được hỗ trợ bởi các **Javascropt Framework** lớn (ví dụ như **AngularJS**, **ReactJS**, BackboneJS, EmberJS,...).

+ Cách tổ chức **code** rõ ràng hơn, hỗ trợ cơ chế giúp kiến trúc hệ thống **code** hướng **module**, giúp xây dựng các hệ thống lớn nơi mà nhiều LTV có thể làm việc cùng nhau một cách dễ dàng hơn.

+ Luôn hỗ trợ, cập nhất các tính năng mới nhất của **JS** (ví dụ như các phiên bản **ECMAScript** mới nhất).

+ Là 1 dự án mã nguồn mở vì vậy nó miễn phí và có cộng đồng hỗ trợ rất lớn.

+ Kiểm tra đc lỗi lúc **compile-time** nên **code** viết bằng **TS** dễ dự đoán và dễ **debug** hơn.

**🡺 Cài đặt TS:**

- Cài đặt gói **NodeJS** (là môi trường sẽ chạy biên dịch **code TS**, tải ở https://nodejs.org/en/).

- Cài đặt **TS**, dùng lệnh: **npm install --global typescript**

**3. ANGULAR OVERVIEW**

- **Angular** – là một **framework** **JavaScript** và được viết bằng **TypeScript**, đc **Google** tạo ra với mục đích viết nên giao diện **web** (**Front-end**) đúng chuẩn “ít nỗ lực”.

Không chỉ đem lại những lợi ích như một **framework**, mà **Angular** còn giữ nguyên cấu trúc như ngôn ngữ lập trình tiêu chuẩn, giúp dễ mở rộng dự án cũng như việc bảo trì.

- **Single page application** (**SPA**) - 1 trang **web** hay 1 ứng dụng **web**, khi tất cả những thao tác xử lý đều được diễn ra trên 1 trang duy nhất.

Các **website** lớn như facebook.com, google.com, youtube.com, twitter.com, … đều được **code** theo kiểu **SPA**. Dễ thấy các **website** kể trên đều có trải nghiệm mượt mà, mà nổi bật nhất chính là **không bị khựng lại** ở thao tác chuyển từ trang này sang trang kia.

- **Kiến trúc Angular:**

+ 1 **Project Angular** sẽ bao gồm nhiều **Module**, mỗi **module** sẽ bao gồm các **Component**, chính là các chức năng của 1 **object**.

+ Các **Component** sẽ ddc hỗ trợ hiển thị nhờ vào **Directive** (vai trò gần giống với **thymeleft**).

+ Các dữ liệu hiển thị ở **Template** **component** đc lấy từ **Service**.

+ **Service** là thành phần kết nối với **back-end** và đc sử dụng trong các **Component** thông qua **DI**.

- **1 số lệnh hay dùng:**

🡺 Cách tạo dự án: ở cửa sổ **terminal** ở thư mục muốn tạo dự án và

gõ lệnh **ng new tên-dự-án --skip-tests** 🡪 chọn [**y**] để tạo **file routing**.

(**--skip-tests** để không tạo **file** **\*.component.spec.ts**, tùy mục đích sử dụng).

🡪 Tạo **component**: **ng g c tên-component** hoặc **ng g c tên-component** **--skip-tests**

**🡪** Tạo **interface**: **ng g i tên-interface**

🡪 Tạo **class**: **ng g class tên-class**

🡪 Chạy dự án: **ng s –o** hoặc click 

🡪 Cài đặt **bootstrap**: **npm i –s bootstrap** hoặc **npm i –s bootstrap@version** (trong đó, **version** là tên phiên bản) hoặc dùng **link CDN** nhúng trực tiếp vào **file index.html**, ví dụ:

<!doctype html>  
<html lang="en">  
<head>  
 <meta charset="utf-8">  
 <title>AngularCaseStudy</title>  
 <base href="/">  
 <meta name="viewport" content="width=device-width, initial-scale=1">  
 <link rel="icon" type="image/x-icon" href="favicon.ico">  
  
 <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.2.0/dist/css/bootstrap.min.css" rel="stylesheet"  
 integrity="sha384-gH2yIJqKdNHPEq0n4Mqa/HGKIhSkIHeL5AyhkYV8i59U5AR6csBvApHHNl/vI1Bx" crossorigin="anonymous">  
 <link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/font-awesome/6.1.2/css/all.min.css">  
</head>  
<body>  
<app-root></app-root>  
  
<script src="https://cdn.jsdelivr.net/npm/bootstrap@5.2.0/dist/js/bootstrap.bundle.min.js"  
 integrity="sha384-A3rJD856KowSb7dwlZdYEkO39Gagi7vIsF0jrRAoQmDKKtQBHUuLZ9AsSv4jD4Xa"  
 crossorigin="anonymous"></script>  
</body>  
</html>

- **Component** – là **thành phần** dùng để xử lý và hiển thị dữ liệu đến với người dùng.

🡺 gồm 3 thành phần chính:

+ **Template** – hiển thị **layout** đến với với người dùng.

+ **Metadata** – dùng để cấu hình cho 1 **component** (**selector**, **templateUrl**,…), liên kết giữa **template** và **class component**.

+ **Class component** – dùng để xử lý dữ liệu.

- **Data binding:**

+ **One way binding** (binding 1 chiều):

- **Interpolation**: **{{ }}** (class component 🡪 template).

- **Property binding**: **[ ]** (class component 🡪 template).

- **Eroperty binding**: **( )** (template 🡪 class component).

+ **Two way binding:** **[(ngModel)]** (binding 2 chiều).

- **Lazy-loading là gì?**

**Lazy Loading** được hiểu nôm na chính là việc **load** dữ liệu khi cần sử dụng đến chúng.

Ví dụ như khi nhiều người cùng vào 1 **page** sẽ không kéo xuống hết đến cuối trang để đọc nội dung thì điều cần làm là **load** nội dung trước. Khi người dùng **scroll** đến đâu dữ liệu sẽ **load** đến đó.

🡪 Là 1 trong những kỹ thuật tối ưu hiệu suất **website** hiệu quả nhất đang được sử dụng phổ biến hiện nay.

🡪 Không mất chi phí băng thông cho những gì không tải xuống, không xem và không nhìn thấy.

🡪 Ví dụ: **Google** tiếp cận với **Lazy loading** theo hướng cụ thể là ở mục tìm kiếm hình ảnh. **Google** sẽ đưa ra danh sách 4-5 bức ảnh liên quan sau khi xem cụ thể một tấm hình nào đó và bên cạnh đó là nút “**View More**” để xem nhiều ảnh hơn.

- **Ưu điểm (tại sao nên sử dụng?) và nhược điểm của Angular:**

🡺 **Ưu điểm:**

+ **Custom Components**: cho phép người dùng xây dựng các **component** của riêng họ và có thể đóng gói chức năng cùng với **logic** kết xuất thành các phần có thể tái sử dụng.

+ **Data Binding**: cho phép người dùng dễ dàng di chuyển dữ liệu, liên kết dữ liệu từ mã **JS** sang **view** và phản hồi với các sự kiện của người dùng.

+ **Dependency Injection**: cho phép người dùng viết các **service** **module** và đưa chúng vào bất cứ nơi nào họ cần, giúp cải thiện khả năng kiểm tra và khả năng tái sử dụng của các **service** giống nhau, giảm sự phụ thuộc giữa các đối tượng.

+ **Testing**: dễ dàng kiểm thử mọi phần trong ứng dụng của mình.

+ **Browser Compatibility**: cung cấp đa nền tảng và tương thích với nhiều trình duyệt. Một ứng dụng **Angular** thường có thể chạy trên tất cả các trình duyệt (**Chrome**, **Firefox**,…) và nhiều hệ điều hành (**Windows**, **macOS**, **Linux**,…).

+ **Comprehensive**: là một **full-fledged Framework** và có thể cung cấp các giải pháp tiện ích cho giao tiếp máy chủ.

🡺 **Nhược điểm:**

+ **Không an toàn**: bản chất của **Angular** là một trong những **Framework Front End**, mà **Front end** này thường vốn không thể bảo mật bằng **Back-end**.

+ Không áp dụng đc cho một số trình duyệt sở hữu tính năng **Disable JavaScript**.

+ Mã ứng dụng **Angular** đc viết bằng **TypeScript**, vì vậy phải có kiến thức nền tảng về **TypeScript**.

- **Angular Cli là gì? Cách sử dụng?**

**- Angular Cli –** là 1 **tool** giúp tạo dự án một cách nhanh chóng. Nó tự động cấu hình các **file** và **package** của dự án **angular** 1 cách tự động. Hỗ trợ tạo các **component**, **directive** hay **service** vào trong dự án có sẵn chỉ bằng 1 dòng lệnh.

- Cách sử dụng:

🡪 Cài đặt **Angular Cli** bằng lệnh: **npm install -g @angular/cli@latest**

🡪 Tạo dự án **angular** bằng lệnh **ng new** …

🡪 Sử dụng các lệnh **ng g** … để tạo các **module**, **component**, **service**, **interface**, **class**,… trong cửa sổ **terminal** của dự án.

- **Cơ chế hoạt động Angular:** **Angular** sẽ làm các bước sau đây để hiển thị trang **home** khi chạy **ng s -o**

🡪 Load file **index.html**.

🡪 Nạp các **thư viện** và các thư viện bên thứ 3 vào.

🡪 Load file **main.ts**.

🡪 Trong file **main.ts** sẽ load module cha là **app.modules.ts**.

🡪 Trong **app.modules.ts** ta load lên **module** cha **component** (**root**) hay còn gọi là **root component**. Trong dự án sẽ có nhiều **component**. Mỗi **component** là 1 phần của **view** hiển thị cho người dùng.

🡪 Trong **module component** sẽ có các file **html**, **css** (**view**) sẽ hiển thị trang **web** cho người dùng.

- **File angular.json và package.js sử dụng để làm gì?**

+ **angular.json** – dùng để cấu hình lại **Angular Cli**.

+ **package.json** –chứa các thư viện cần thiết cho dự án **angular**, ngoài ra nếu ta thêm một thư viện bên thứ 3 vào thì khai báo trong này.

- **Biên dịch AOT là gì? Những lợi thế của AOT?**

+ **Ahead-of-Time compilation** (**AOT**) – là khái niệm liên quan nhiều hơn tới việc cải thiện hiệu năng **run-time**, có thể biên dịch **TypeScript** trong lúc **run-time**.

+ Những lợi thế:

- Kết xuất nhanh hơn.

- Ít yêu cầu không đồng bộ hơn.

- Kích thước tải xuống khung nhỏ hơn.

- Phát hiện lỗi mẫu trước đó.

- An ninh, bảo mật tốt hơn.

- **Angular đi theo mô hình nào? Phân biệt mô hình MVVM và MVC?**

**AngularJS** đc ra mắt vào **20/10/2010**, do lập trình viên **Misko Hevery** tại **Google** viết ra như là một dự án kiểu “viết cho vui”. Lúc đó **AngularJS** được viết theo mô hình **MVC**. Nhưng sau này, **AngularJS** không còn triển khai **MVC** theo cách truyền thống, mà gắn liền hơn với **Model-View-ViewModel**.

🡺 **Mô hình MVC:**

- **Model** là thành phần trung tâm thể hiện hành vi của ứng dụng và quản lí dữ liệu.

- **View** được tạo ra dựa trên thông tin của **Model** .

- **Controller** đóng vai trò trung gian giữa **Model** và **View** và để xử lý logic .

🡺 **Mô hình MVVM:**

- **View**: tương tự mô hình **MVC**, **View** là phần giao diện của ứng dụng để hiển thị dữ liệu và nhận tương tác của người dùng.

Điểm khác biệt là **View** trong mô hình này tích cực hơn. Nó có khả năng thực hiện các hành vi và phản hồi lại người dùng thông qua tính năng **binding**, **command**.

- **Model**: tương tự như mô hình **MVC**, **Model** là các đối tượng giúp truy xuất và thao tác trên dữ liệu thực sự.

- **ViewModel**: Lớp trung gian giữa **View** và **Model**. **ViewModel** có thể được xem là thành phần thay thế cho **Controller** trong mô hình **MVC**. Nó chứa các mã lệnh cần thiết để thực hiện **data binding**, **command**.

**ViewModel** không hề biết gì về **View**, một **ViewModel** có thể được sử dụng cho nhiều **View** (one-to- many).

**4. ANGULAR COMPONENT & TEMPLATE**

- **Truyền dữ liệu từ component cha 🡪 component con:**

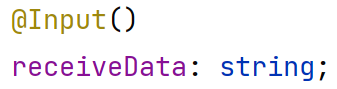
+ Sử dụng **@Input**, khai báo thuộc tính trong **class component con**.

+ Ví dụ:

🡪 Khai báo 1 biến ở **class component cha** (ở parent.component.ts).



🡪 Sử dụng **@Input**, khai báo thuộc tính trong **class component con** (ở child.component.ts).



🡪 Nhúng **component con** vào **component cha** thông qua thẻ **selector** và truyền giá trị thông qua thuộc tính (ở parent.component.html).

🡪 Hiển thị kết quả ở **component con** (ở child.component.html).

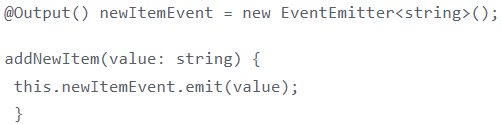
<p>Result: **{{receiveData}}**</p>

- **Truyền dữ liệu từ component con 🡪 component cha:**

+ Sử dụng **@Output + EvenEmitter** trong **class component con**.

+ Ví dụ:

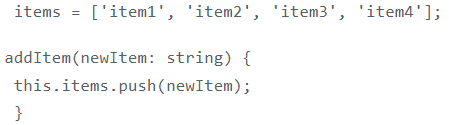
🡪 Ở **component con** khai báo + đánh dấu thuộc tính **newItemEvent** sẽ đc gửi sang **component cha** và tạo phương thức **addNewItem()** (ở child.component.ts).



🡪 Tạo **file html** cho **component con** (ở child.component.html).



🡪 Cấu hình **component cha** (ở parent.component.ts).



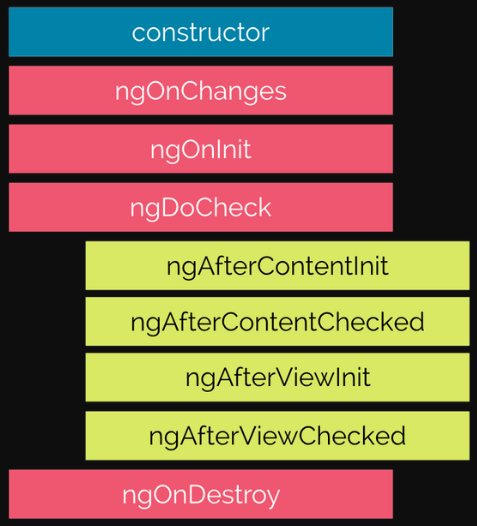
🡪 Nhúng **component con** vào **component cha** thông qua thẻ **selector** (ở parent.component.html).

<app-item-output **(newItemEvent)="addItem($event)"**></app-item-output>

+ Có thể sử dụng **@ViewChild**.

+ Ngoài ra, có thể sử dụng **Service** đối với 2 **component** không quan hệ.

- **Vòng đời component:**



🡪 **Constructor** – sẽ hoạt động đầu tiên, và thường sử dụng cho **DI** các thuộc tính vào **component**.

(**Constructor** là khái niệm có trước, không nằm trong vòng đời của **component**).

🡪 **ngOnChange()** – khi xảy ra **Input** (truyền dữ liệu từ **component** cha 🡪 con).

🡪 **ngOnInit()** – khởi tạo các dữ liệu ở **component** (khi **component** bắt đầu **load** trên **layout**).

🡪 **ngDoCheck()** – xảy ra khi có sự thay đổi thông tin hoặc có sự kiện ở **component**.

🡪 **ngOnDestroy()** – xảy ra khi **component** không còn hiển thị ở **layout**.

- **Directive** – dùng để thay đổi cấu trúc ở **template**, thao tác với các thành phần giao diện (thao tác với các **element** của **DOM**).

🡺 Gồm 4 loại:

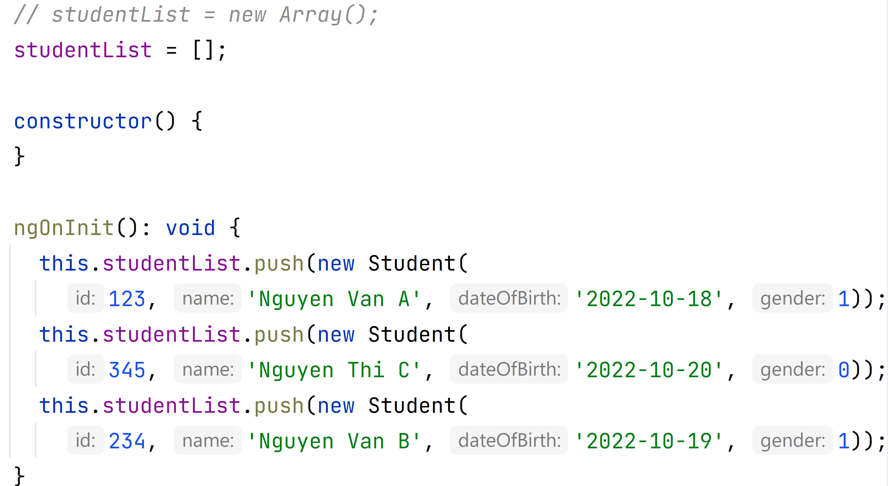
+ **Component directive** – chính là thẻ **component** đc tạo từ **selector**.

+ **Structural directive** – gồm \***ngFor**, \***ngIf**,…

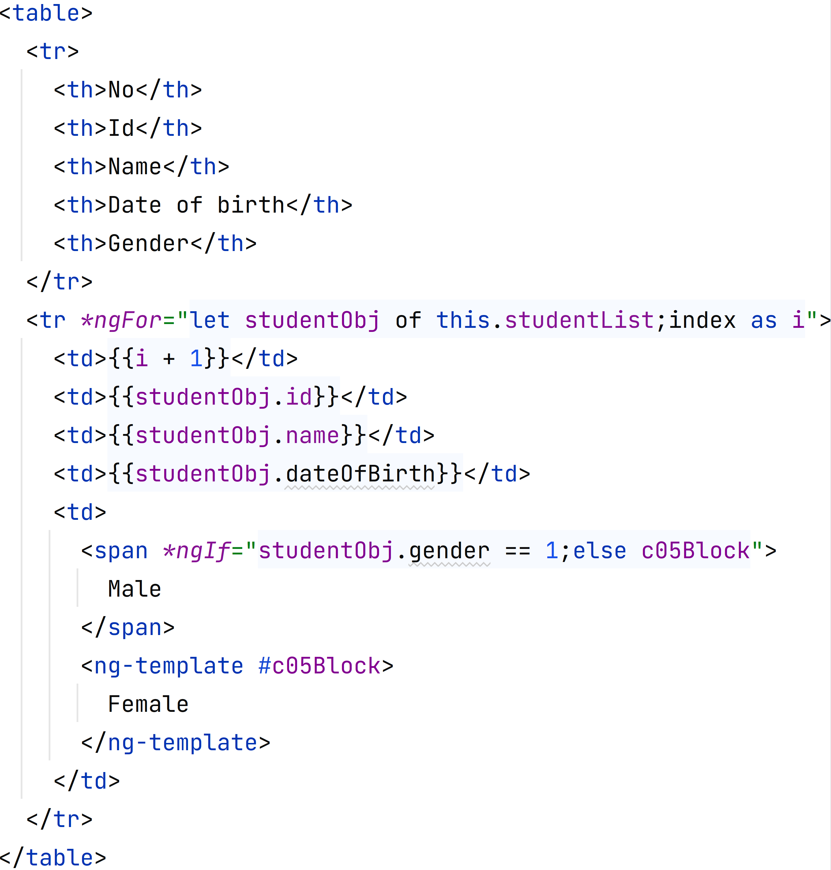
**Ví dụ:**

🡪 Tạo class **Student**.

🡪 Import class **Student** và thêm dữ liệu cứng ở file **\*.component.ts**.



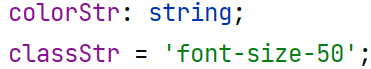
🡪 Sử dụng \***ngFor**, \***ngIf**,… ở file **\*.component.html**.



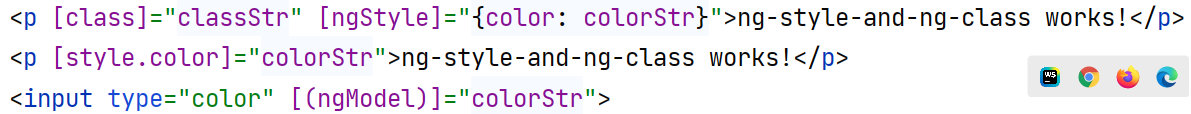
+ **Attribute directive** – gồm **ngStyle**, **ngClass**,…

**Ví dụ:**

🡪 Khai báo thuộc tính ở file **\*.component.ts**.



🡪 Sử dụng **ngStyle**, **ngClass**,… ở file **\*.component.html** (lưu ý: **[ngStyle]**, **[ngClass]** có thể áp dụng cho nhiều đặc tính của **style / class**, sử dụng linh động, còn **[Style]**, **[Class]** chỉ áp dụng cho 1 đặc tính duy nhất).



+ **Custom directive** – sẽ có những trường hợp những **Directive** trên không phù hợp với mục đích bài toán của mình, nên trong **Angular** cho phép chúng ta tự viết ra một **directive** riêng cho mình và sử dụng theo cách mình muốn.

- **Pipe** – dùng để định dạng (**format**) dữ liệu, chuyển đổi dữ liệu hiển thị lên **template** cho người dùng có thể hiểu được.

🡪 Đc sử dụng ở cả **template** và **class component** (**DI** trong **constructor**).

🡺Ví dụ: **titleCase** (viết hoa chữ cái đầu), **currency** (định dạng lại số hay đơn vị tiền), **upperCase** (in hoa toàn bộ),…

**5. ANGULAR FORM**

- **Phân biệt Template Driven Form & Reactive form:**

+ Giống: đều là phương tiện cho người dùng nhập liệu và gửi đi.

+ Khác:

|  |  |  |
| --- | --- | --- |
|  | **Reactive form** | **Template Driven Form** |
| Cách tạo | - Tạo **form** ở **class component**, sau đó **binding** sang **template** (thông qua **formGroup**, **formControl**, **formArray**,..). | - Tạo ở **template** dựa trên các **directive** (bằng cơ chế **two way dinding**). |
| Đồng bộ dữ liệu | - Có sự đồng bộ dữ liệu giữa **form** ở **template** và **form** ở **class component**. | - Chỉ có **form** ở phía **template**, ở **class** thì không, dẫn đến sự không đồng bộ dữ liệu ở 2 phía. |
| Validate | - Thực hiện ở **class**, dựa trên các **function** ở **Angular** (dễ dàng hơn, việc **validate** xử lý đa dạng hơn, có thể **custom**). | - Thực hiện ở **template**, dựa trên các **directive** ở **HTML** (khó khăn trong việc **custom validate**, xử lý không triệt để). |
| Tính bất biến | - Không làm thay đổi trực tiếp **data model**. | - Vì có sự **binding** 2 chiều dữ liệu nên sẽ làm thay đổi trực tiếp các thuộc tính ở **data model**. |
| Khác.. | - Vì được viết ở **class component** nên linh hoạt và dễ kiểm thử. | - Không linh hoạt. |

- **Validate:**

+ Sử dụng các **validators** có sẵn (required, pattern, email, min, max, minLength, maxLength,…).

+ 3 trạng thái xảy ra ở input:

- **pristine** – chưa tương tác cũng sẽ báo lỗi.

- **touched** – chạm vào và sẽ báo lỗi ngay khi click ra khỏi ô input.

- **dirty** – nhập giá trị vào nếu sai sẽ báo lỗi.

- **Custom validate:** Tạo 1 **function**:

+ Đầu vào (input): **abstractControl** (có thể là **formGroup**, **formControl**, **formArray**,…).

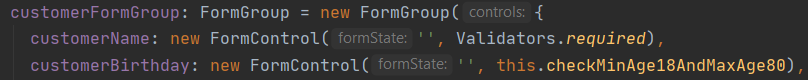
+ Đầu ra (output):

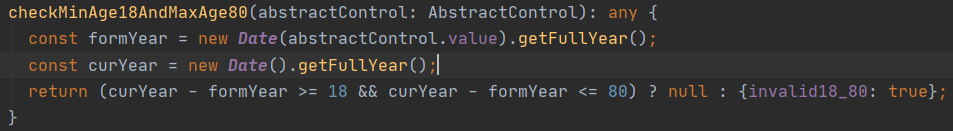
- Đúng: trả về **null**

- Sai: trả về **{errorCode: true}**

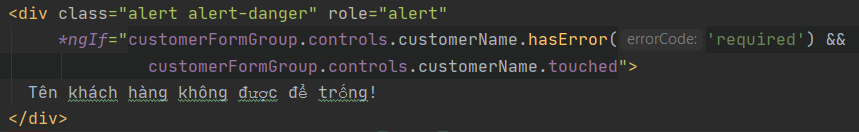
🡺 Ví dụ:

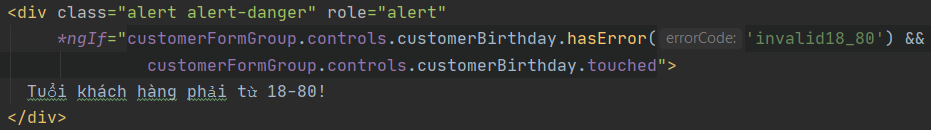
🡪 **Validate** ở **class component:**





🡪 Hiển thị lỗi ở **template:**





**6. SERVICE & ROUTER**

- **Service:**

+ Mục đích:

- Dùng để xử lý các nghiệp vụ bị lặp lại, tái sử dụng **code**, thuận tiện cho việc bảo trì **code**.

- Dùng để ghi **log**, **call** **API**,…

+ Cách tạo: **ng generate service tên-service** hoặc **ng g s tên-service**.

- **Router** – là một **module** được đặt tại **@angular/router**, cung cấp cho ứng dụng **Angluar** khả năng điều hướng, thay đổi giao diện tương ứng với từng **component** dựa theo **URL**.

+ Cách tạo: tạo ngay từ đầu khi tạo dự án hoặc dùng lệnh **ng g m tên-router --flat -module=app**

+ Một số khái niệm:

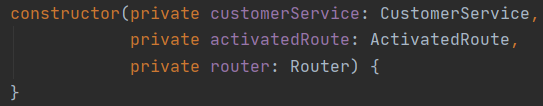
- **Routers** – khai báo **URL** cho **component** ở **file routing**.

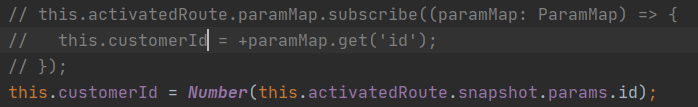
- **<router-outlet></router-outlet>** – thẻ hiển thị **layout** của các **component** tương ứng với các **URL**.

- **routerLink** – là 1 **directive** của **Angular**, giúp cập nhật **URL** ứng với **component** mà ta muốn hiển thị trên **layout** (tương tự như **href=””**).

- **Router** – dùng để thay đổi **component** dựa theo **URL** tương ứng đã khai báo (ví dụ như sau khi thêm mới / chỉnh sửa, ta muốn hiển thị lại giao diện danh sách ban đầu thì sử dụng **Router**).

- **ActivatedRoute** – dùng để lấy tham số ở **URL** (phải giống với tham số đã khai báo ở **Routing**).





- **Trình bày sự khác nhau giữa navigateByUrl và navigate?**

+ **navigateByUrl** – là phương thức tương tự như thay đổi trực tiếp thanh vị trí, cung cấp 1 **URL** mới “hoàn toàn”, được sử dụng để điều hướng đường dẫn tuyệt đối.

Ví dụ: muốn điều hướng từ abc.com/user đến abc.com/assets , trong trường hợp này có thể sử dụng **router.navigateByUrl ()**.

+ **navigate** – là phương thức cung cấp đường dẫn tương đối, điều hướng tương đối đến đường dẫn hiện tại.

Ví dụ: nếu đường dẫn hiện tại là abc.com/user , muốn điều hướng đến abc.com/user/10 thì có thể sử dụng **router.navigate()**.

**7. CONNECT TO BACK-END**

- **HttpClient** – dùng để hỗ trợ trong việc kết nối với **Back-end** (được khai báo ở **service**).

+ Cần khai báo **HttpClientModule** ở **file app-module.ts**.

+ Gồm 5 phương thức chính:

- **GET** – dùng để lấy thông tin đối tượng.

- **POST** – dùng để đăng ký mới thông tin đối tượng.

- **PUT** – dùng để cập nhập toàn bộ thông tin đối tượng (phải gửi 1 bản ghi đầy đủ các trường để yêu cầu cập nhật, nếu không truyền đầy đủ dữ liệu thì các dữ liệu đó sẽ đc **update** với giá trị mặc định).

- **PATCH** – dùng để cập nhật đúng các thông tin truyền vào (giống với **PUT** về ý nghĩa nhưng khác nhau về cách làm việc, nó chỉ thay đổi những trường đc yêu cầu thay vì thay đổi toàn bộ bản ghi).

- **DELETE** – dùng để xóa thông tin đối tượng.

- **Observable** – là thành phần dùng để quản lý các luồng dữ liệu từ **web service** truyền vào.

- **Subscribe** – là phương thức để đăng kí lắng nghe các sự kiện từ **Observable**, và nhận vào 1 **Observer**.

- **Observer** – là cách thức xử lý 3 loại trạng thái mà một **Observable** có thể gửi:

+ **next** – bắt buộc phải có, được sử dụng để lấy ra dữ liệu từ **Observable**.

+ **error** – không bắt buộc phải có, được sử dụng để xử lý các lỗi xảy ra trong quá trình xử lý.

+ **complete** – không bắt buộc phải có, đc sử dụng để xử lý các nội dung sau khi một **Observers** hoàn thành, dù có lỗi hay không (gần giống với **finally**).

- **So sánh** **Observable và Promise:**

+ Giống:

- Xử lý đc bất đồng bộ.

- **Observable** cũng có những tính năng của **Promise** và thêm một số ưu điểm khác.

+ Khác:

|  |  |
| --- | --- |
| **Observable** | **Promise** |
| - Trả về một hoặc nhiều giá trị.  - Có thể **cancel request**.  - Chỉ được khởi tạo khỉ và chỉ khi có đăng ký đến **observable** đó (hỗ trợ xử lý **bất đồng bộ** tốt hơn).  - Có thể **retry** (ví dụ như việc kết nối đến **web service** gặp lỗi, ta có thể sử dụng **retry** để thử kết nối lại bao nhiều lần tùy thích).  - Có thể dùng với **event** (thao tác hướng sự kiện).  - Có các thao tác **tiền xử lý dữ liệu** như: filter, map, … | - Chỉ trả về một giá trị duy nhất, đó có thể là object, array, number,…  - Không thể **cancel** được **request**.  - Khi khai báo sẻ đc khởi tạo ngay lập tức mặc dù chưa có bất kỳ đăng ký nào. |

**8. MODULE & DI**

- **Module** – dùng để phân chia **project** thành nhiều **sub-project**, mỗi **sub-project** đc gọi là 1 **module**, mỗi **module** sẽ bao gồm các thành phần như 1 **project** thông thường.

🡪 Là nơi nhóm các **component**, **directive**, **pipe** và **service** có liên quan đến ứng dụng **Angular**.

🡪 Việc phân chia hợp lí, khoa học sẽ giúp dự án phát triển dễ dàng, dễ bảo trì, dễ tiếp cận, hỗ trợ làm việc nhóm,…

🡺 Cách tạo Module: **ng g m tên-module --routing**

- **Các thành phần của module:**

+ **declarations** – dùng để khai báo các **component**, **directive**, **pipe** phụ thuộc vào **module** (là các thành phần sẽ dùng ở trên **template**).

+ **imports** – dùng để khai báo các **module** khác đc sử dụng trong ứng dụng.

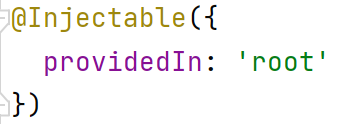
+ **providers** – dùng để **inject** các **service** vào **module**.

+ **bootstrap** – khai báo **component** dùng để khởi động **project** (định nghĩa **component** gốc của **module**).

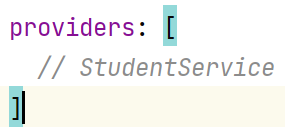
- **Dependency Injection** (**DI**) – là **inject** các đối tượng này vào trong các đối tượng khác, nhằm giảm sự phụ thuộc, sự kết dính giữa các đối tượng với nhau.

🡺 Phạm vi và cách thực hiện:

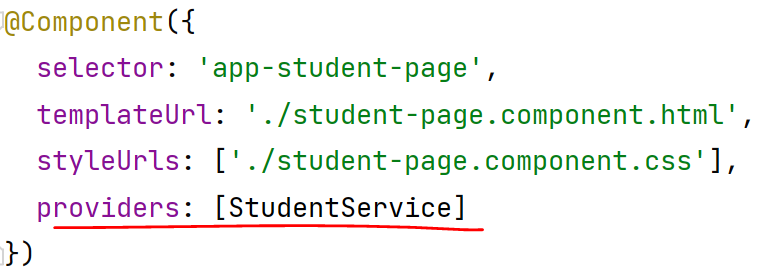
**+ DI cho toàn bộ dự án**: trong **service**, khai báo:



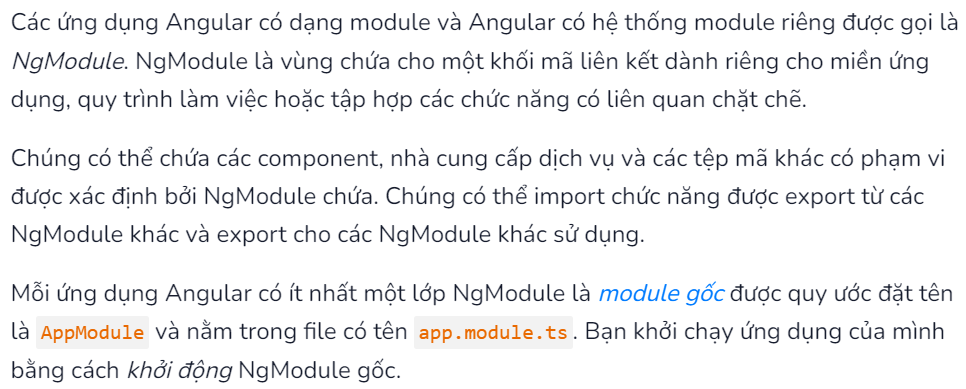
**+** **DI cho toàn bộ module**: khai báo ở **providers** của **@ngModule** (ở file **\*.module.ts**).

Vd: 

**+ DI cho toàn bộ component**: khai báo ở **providers** của **@Component** (ở file **\*.component.ts**).

Vd: 

🡺 **ngModule là gì?**



\_**The Last Shinobi ϟ\_**