Giáo án - lộ trình

[**Thông tin cơ bản**](#_ql54ti9bng7d) **5**

[Overview:](#_7d04lhhhz7s7) 5

[Tài liệu cần đọc:](#_mea3kg6bfbvv) 5

[Đối tượng:](#_9ytps0mvm3ik) 5

[Lộ trình (4 Section)](#_u0n84cttgxi8) 6

[**Giáo án**](#_t3dphpdqcbm6) **6**

[Section 1](#_lusu5yo18rm2) 6

[Nodejs](#_ffsfpf4w1akb) 6

[Introduction](#_6nnz9gqvbxgt) 6

[JS Advantive](#_l3jqg25s4jwz) 6

[Deep down js](#_rgayroh87lve) 6

[Callback](#_k1bwndf7hqr7) 6

[Class](#_fhhhrw69wilu) 7

[Try catch](#_2rktxfxbdxuo) 8

[This](#_dmvozx1e45q0) 9

[Closures](#_5i3f4h8hwnmn) 9

[Bind](#_ysjpnz3f7hfi) 11

[Call - apply](#_ygn9rgad3wom) 12

[Array function](#_eksdg0t2ut1b) 14

[Map](#_c87wdqnra3zs) 14

[Filter](#_w0drnvlwqgjp) 15

[Find](#_y5xns09548bs) 15

[Reduce](#_92g93wim1xec) 16

[Sort](#_x9nar73mjhcx) 17

[Pop](#_z6dfmkvp9w1m) 17

[Push](#_vx8krzus8ckt) 18

[\* Bài tập](#_agzlbax1cu7v) 18

[Oop](#_a01hx2oxl9bt) 23

[Es6](#_4tr64bgaimei) 23

[Block scope](#_igpaktgchzhw) 23

[Arrow function](#_d06tyw7q4ll1) 24

[Rest parameters](#_l2dgz6h32ie3) 26

[Spread syntax](#_qv373n5rqbcf) 27

[Destructuring](#_hawrwjjlf17v) 30

[Template string](#_umk974jit33o) 31

[Promise](#_jkbrngmjgkl1) 35

[Async - await](#_oh78m18mfljy) 37

[Vậy khi có Async / Await có làm cho promises lỗi thời?](#_dv1c22soob0u) 38

[Iterator - iterable](#_6ndiioo9tlmg) 39

[Express](#_133mj3dq7e29) 40

[Expressjs là gì?](#_iwnlaqt4gn9y) 40

[Setup](#_bwvbgl84kptb) 41

[Chạy chương trình đầu tiên](#_8u1ifnfsd2sn) 41

[Routing trong express](#_7tasnx57hh9d) 42

[\* Bài tập](#_lpa4x0ikqrfz) 44

[Section 2:](#_3igoi9nvwpdh) 44

[Typescript](#_edwo2lat8uma) 44

[Setup](#_hjuuufecg8g5) 44

[Cài đặt môi trường](#_yaiues4knud9) 44

[Chạy code](#_4bma82yszdgn) 44

[Tsconfig.json](#_f3zavfj1c2qz) 44

[Type](#_915eltew1q43) 45

[Basics type](#_n8stzfl7un3l) 45

[Function](#_swatohdlr1k4) 47

[Union Types](#_683oi31h8wzn) 48

[Interfaces](#_zgjcy391acd1) 49

[Generics](#_nfvxbdo136g) 49

[Một số Utility types phổ biến](#_6slj4y4bqwmh) 50

[Express](#_bfradfjz97sx) 52

[Cơ bản:](#_w2mirrr2z241) 52

[Deep down:](#_w0bvjjr3wosu) 52

[Module import](#_4h0vg45lz7vo) 52

[Package json](#_jrwr5g69obwr) 54

[Npm, npx, nvm](#_qslavkrela54) 56

[NPM](#_9ro1czvb7zd2) 56

[NPX](#_9llrfijxqvxs) 56

[NVM](#_30llwjgx3ppk) 57

[Middleware](#_73tak7xg1pvt) 57

[Định nghĩa](#_hkm23o4ifdx3) 57

[Application-level middleware](#_79e0m4jmhxlk) 58

[Router-level middleware](#_z5yoq8whz4j6) 59

[Error-handling middleware](#_6ujxea5px5nd) 60

[Built-in middleware](#_5rx8zp1neqlf) 60

[Third-party middleware](#_koaog3f7ywgr) 61

[Session](#_1fqedpiwab5f) 61

[Định nghĩa](#_vfatt2w36362) 61

[Session trong Express](#_6sljdbwu50tf) 61

[Auth](#_l1510zz2xhix) 63

[JWT](#_ke9nzmr5du9j) 63

[Authentication](#_8x0lkrr67pif) 64

[Authorization](#_azazd7d5mjrq) 65

[Midterm project](#_m8a1h9iuge5r) 65

[Dự án task/issue checking:](#_7pns6e3v0xc8) 65

[Section 3](#_fhqlzob7fr9n) 67

[DB connection](#_4fszctaltm82) 68

[Tích hợp TypeOrm](#_6dprnnhs8rkk) 68

[Repository pattern](#_d6qbt2tgw0q4) 69

[Relations](#_lmkktskn6cr) 70

[Transactions](#_rhzmlgdvgys) 71

[Migrations](#_z5ro7e9rnvz) 72

[Multiple databases](#_cn6n4di37igj) 73

[Tích hợp Sequelize](#_gu7u8it00060) 75

[Cài đặt](#_5mhuxk3qj6jp) 75

[Xây dựng model cho các table](#_o6uj8g38rkiq) 76

[Xây dựng quan hệ các model](#_tslnrcu3mkhj) 82

[Tự động load các model](#_irpymfbn1aiu) 83

[Transaction](#_hm4vm28zgx1d) 84

[Kết nối bất đồng bộ](#_3uityc12pzcy) 87

[Nestjs](#_4y0vugqunonr) 88

[Định nghĩa](#_i807t3w1o9bj) 88

[Setup](#_lbc6gx1o3tc3) 88

[Controller](#_i559l3w9zeq2) 89

[Routing](#_eefivz5jm52t) 89

[Thông tin Request](#_ryym8e86e8n7) 91

[HTTP methods](#_vi4rprtbxrt8) 92

[HTTP Status Code](#_gibr42nezaxn) 92

[Headers](#_1athlctzoook) 93

[Redirect](#_27j7899vncwn) 94

[Route parameters](#_48v93l5pe4mp) 95

[Request scopes](#_m775zkizwoe8) 95

[Bất đồng bộ (Asynchronicity)](#_pbbrixlebew) 95

[Truyền nhận dữ liệu qua DTO](#_5v75lrcinq4v) 96

[Provider](#_5tvsb8jaiwei) 98

[Dependency Injection](#_kuw5o75pjpsg) 98

[Providers customize](#_w0z76enfce09) 100

[Non-class provider tokens](#_bgk2b3f26wvx) 103

[Export Provider](#_uws67b2bar0a) 104

[Module](#_mqxqqww56hmx) 105

[Shared modules](#_7nl9v91xz1w6) 105

[Module re-exporting](#_md3os14n0jrg) 106

[Dependency injection](#_bud9wcv3sozn) 106

[Global modules](#_rnwut85h9eme) 107

[Dynamic modules](#_pcifwfhlpfb1) 108

[Middleware](#_bdg04jeil2ca) 110

[Filter](#_1n5op4pnek8j) 112

[Pipe : link](#_wbdw074wroxv) 114

[Guard: link](#_48avqlx8hdzr) 114

[Interceptor: link](#_62e5q44i0vv8) 114

[Custom decorator: link](#_mcoghy9sc408) 114

[Dynamic module](#_ijhv9pd5q5l1) 114

[Dynamic module use case](#_ehagrs5qhzao) 116

[Circular dependency](#_oko1w8f3311) 118

[Module reference : link](#_2oh9fd8nnncb) 118

[Lifecircle event](#_9h2waf4zcsn9) 118

[Testing](#_52wdqb9gz4bp) 120

[Technical](#_nejm22gewlrj) 120

[Database](#_afkmthem6z7h) 120

[Mongo](#_y1l2in39824) 120

[Configuration](#_c92dc3n04ft3) 121

[Validation](#_ep7yt6xbvxw6) 121

[Caching](#_b3joqohg8gqd) 121

[Serialization](#_37kgi71pc1wn) 121

[Task Schedule](#_xrobnk53khjc) 122

[Queues](#_o56mbia7h38g) 123

[Cookies](#_o8i9ae802d6t) 124

[File upload](#_v7jmdzn2ugj2) 125

[Streaming file](#_xuhey2cc2lv) 125

[Axios](#_ov48f5ggyuib) 126

[Session](#_lj9i5dh5a3i0) 126

[Section 4](#_g7jqhkke3wv1) 126

[Nestjs](#_cn7p70428jvy) 126

[Final project](#_q0pg0rkgksu7) 126

# Thông tin cơ bản

## Overview:

Giáo trình training intern nodejs cho các bạn mới biết code.

## Tài liệu cần đọc:

## Đối tượng:

Đối tượng 1: Intern chưa tiếp xúc với node nhưng biết js và tư duy tốt

* Tư duy tốt
* Đã nắm rõ kiến thức js cơ bản:
  + Biến (var, let, const)
  + Data type (primitive values, object,...)
  + Toán tử
  + Hàm
  + Vòng lặp
  + Condition
  + Array, object
  + Callback
  + Oop (optional)
* Cơ sở dữ liệu
  + Tối thiểu 1 loại sql
  + Relation
  + Join
* Nodejs cơ bản (optional)
* Http request (optional)

Đối tượng 2: Tự học nodejs có dự án cá nhân ổn

* Đáp ứng được điều kiện của đối tượng 1
* 1 project cá nhân
* Js Advance
* Database
  + Index (optional)
  + Transaction (optional)
* Nodejs - Express
  + Event loop
  + Callback - callback hell
  + Client Request
  + Authen
  + Kết nối DB (dùng gì - curd)
  + Socket (optional)
  + Session cookies

Đối tượng 3: Cuối intern

* Đáp ứng được điều kiện của đối tượng 2
* Nắm chắc về restful
* Typescript (optional)
* Đã tiếp xúc với dự án thật hoặc dự án cá nhân phức tạp

## Lộ trình (4 Section)

* Section 1: Nodejs (tool - js advance) + Express
* Section 2: Express + Typescript + <Midterm project>
* Section 3: DB connection + nestjs
* Section 4: Nestjs + <Final project>

# Giáo án

## Section 1

### Nodejs

#### Introduction

* Nodejs
* Event loop
* Setup
* Vs code + extension
* Coding convention

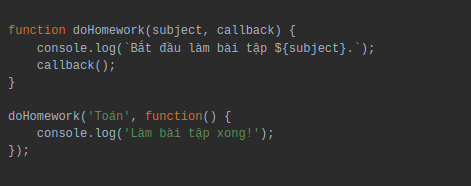
#### JS Advantive

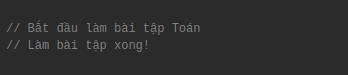
##### Deep down js

###### Callback

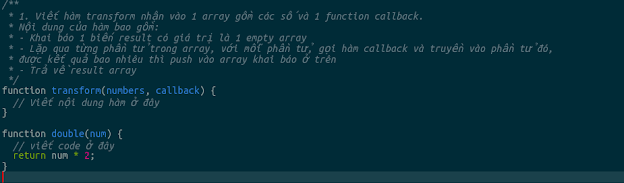
* Callback là một hàm sẽ được thực hiện sau khi một hàm khác đã thực hiện xong
* Callback được sử dụng như một đối số truyền vào

VD:



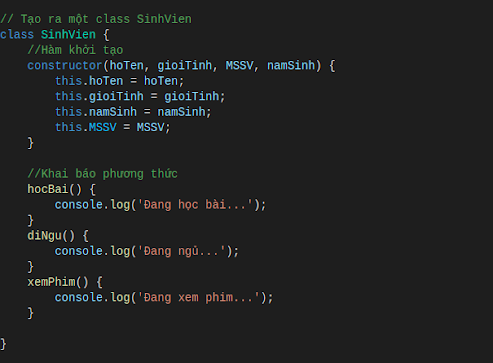


Bài tập

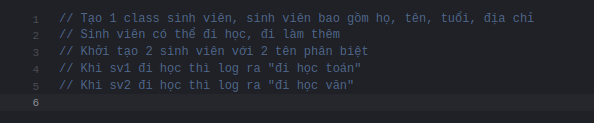


###### Class

* Trong lập trình hướng đối tượng class (hay lớp) được sử dụng để tạo ra các đối tượng
* Đối tượng thì sẽ có các thuộc tính, phương thức.
* Các đối tượng giống nhau sẽ có những thuộc tính và phương thức giống nhau.

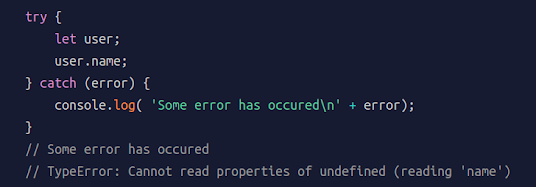


Bài tập



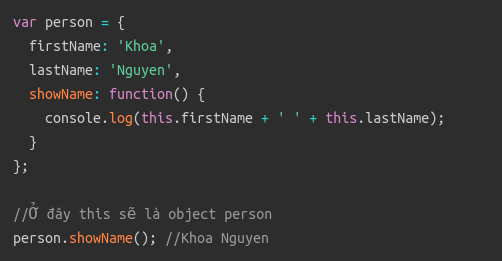
###### Try catch

* Try catch là một khối lệnh dùng để bắt lỗi chương trình trong javascript. Ta sử dụng try catch khi muốn chương trình không bị dừng khi một lệnh nào đó bị lỗi. Thường thì đó là những lỗi do người dùng nhập sai dữ liệu, hoặc người dùng thao tác bị sai.

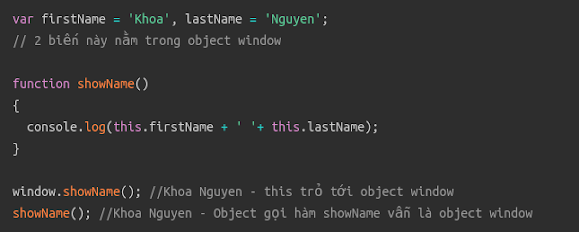


###### This

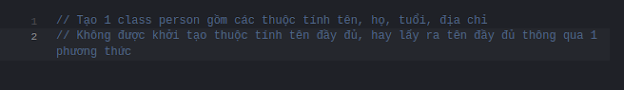
* Trong javascript, chúng ta dùng từ khóa this để đại diện cho một đối tượng (Object). Đối tượng đó là chủ thế của ngữ cảnh, hoặc là chủ thế của code đang được chạy.



* Một trường hợp khác, khi ta khai báo biến global và hàm global, toàn bộ các biến và hàm đó sẽ nằm trong một object có tên là window. Lúc này, khi ta gọi hàm showName, chính object window là object gọi hàm đó, this trỏ tới chính object window.



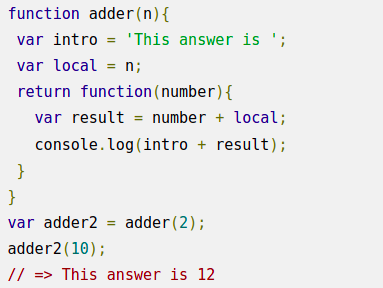
Bài tập



###### Closures

* Một closure là một hàm bên trong mà có thể truy cập biến của hàm bên ngoài (chứa nó)

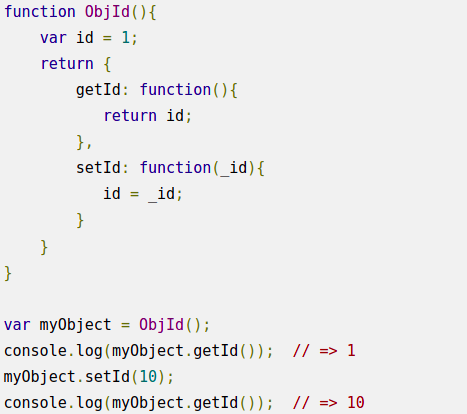
VD:



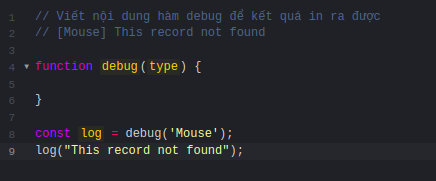
=> Khi mình hàm adder(2), hàm số này thực hiện và kết quả trả về được gán vào biến adder2. Khi gọi adder2(10), kết quả trả về là 12, chứng tỏ hàm closures vẫn có thể truy cập tới biến cục bộ của outer function khi hàm đó thực hiện

xong.

* Hàm closures lưu trữ biến số của outer function theo kiểu tham chiếu



Bài tập

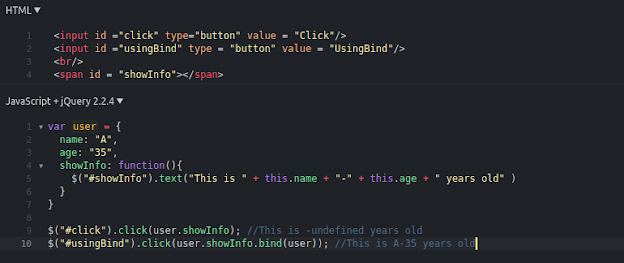


###### Bind

bind() cho phép chúng ta dễ dàng thiết lập một đối tượng cụ thể sẽ bị ràng buộc này khi một chức năng hoặc phương thức được gọi.

Các tác dụng của bind:

+) Cho phép chúng ta set giá trị của biến "this"



Mục đích trong ví dụ này ta sẽ xem xet bind có thể set giá trị “this” trong object như nào. Ta có 1 object user chứa function showInfo có nhiệm vụ hiển thị thông tin của object user đó thông qua biến “this”

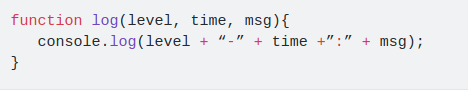
Ở trường hợp đầu tiên khi không sử dụng bind ta có thể thấy khi click vào button giá trị “this” trong hàm showInfo không được gọi ( bằng “undefined”).

Ở trường hợp thứ 2 khi ta dùng bind để set giá trị this của user. Lúc này ta có thể bind giá trị của user với this. Kết quả ta thấy show được thông tin của user rồi đó.

+) Cho phép tạo một Curry Function

Không chỉ bind được giá trị “this” , hàm bind còn có thể bind được các tham số truyền vào cho hàm. Tức là ta có thể tạo ra một function mới từ function cũ bằng cách gắn mặc định một tham số cho function cũ đó . Đó là cách tạo một Curry Function

Giả sử mình có một hàm log đơn giản có 3 tham số truyền vào như sau:



Bây giờ mình muốn tạo một hàm log access ngày hôm nay mình sẽ tạo ra sử dụng hàm cũ như sau:

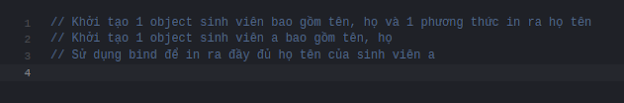


Ta có thể tạo function mới bằng cách sử dụng bind bằng cách gán mặc định 2 tham số đầu tiên như sau:



Khi gọi logAccessToday(“Server Access”) đều ra một thông báo như nhau.

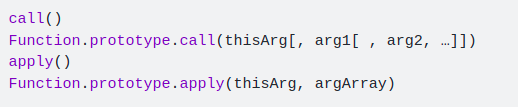
bài tập

Call - apply

- Chức năng chung của chúng là xác định một tham số, xác định this và truyền các tham số còn lại vào.

- Điểm khác nhau cơ bản giữa chúng là apply sẽ truyền 1 array các tham số còn call sẽ truyền lần lượt các tham số.

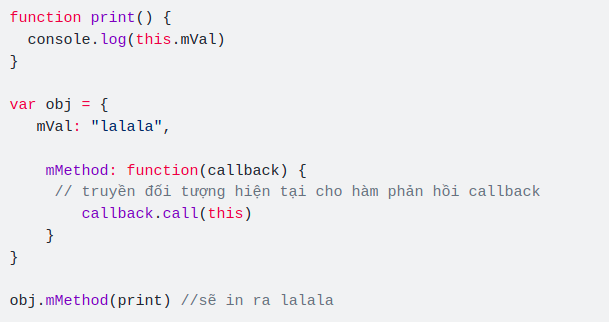
Cú pháp :



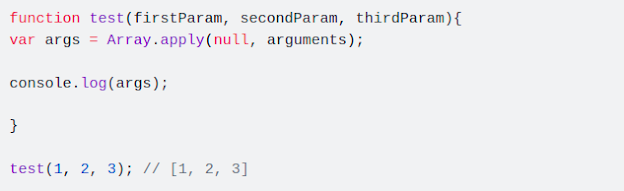
Ta sẽ cùng xem xet ví dụ sau:



Sử dụng call , apply để set this cho hàm callback



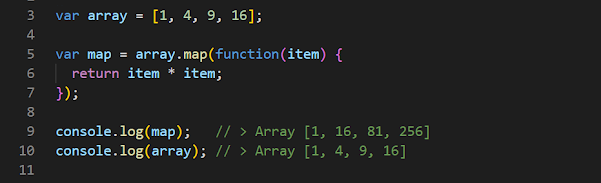
Sử dụng để mượn hàm (borrowing function)



##### Array function

###### Map

* Hàm map() có chức năng tương tự như vòng lặp forEach.
* Hàm map() giúp tạo ra một mảng mới với các phần tử là kết quả từ việc thực thi một hàm lên từng phần tử của mảng ban đầu
* map() không làm thay đổi giá trị của mảng

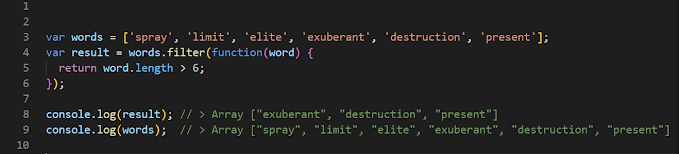


* Cú pháp

array.map(function(currentValue, index, arr), thisValue)

###### Filter

* Filter dùng để lọc các phần tử trong mảng thỏa mãn 1 điều kiện nào đó
* filter() không làm thay đổi mảng ban đầu
* filter() trả về 1 mảng mới sau khi lọc
* filter() trả về mảng rỗng nếu không có phần tử nào thỏa mãn điều kiện
* VD:

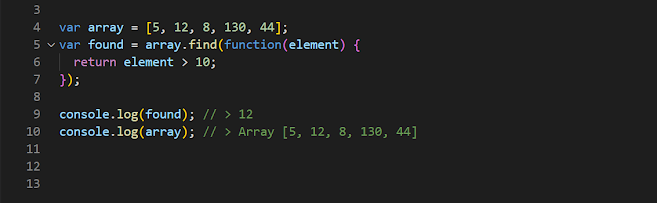


* Cú pháp

array.filter(function(currentValue, index, arr), thisValue)

###### Find

* Dùng để lọc phần tử trong mảng và trả về giá trị đầu tiên tìm thấy trong mảng hoặc undefined nếu không tìm thấy.
* find() không làm thay đổi mảng ban đầu
* VD:

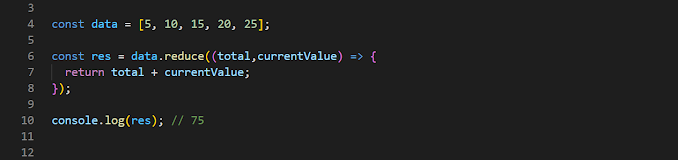


* Cú pháp

array.find(function(currentValue, index, arr),thisValue)

###### Reduce

* reduce() dùng để thực thi một hàm lên từng phần tử của mảng ( trừ trái sang phải ) với một biến tích lũy để thu về một giá trị duy nhất
* reduce() không làm thay đổi mảng ban đầu
* reduce() trả về giá trị sau khi rút gọn
* VD:



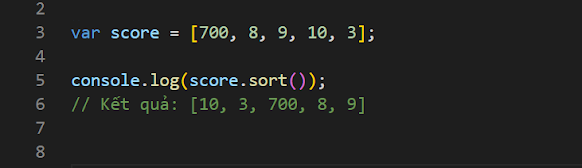
* Cú pháp

array.reduce(function(total, currentValue, currentIndex, arr), initialValue)

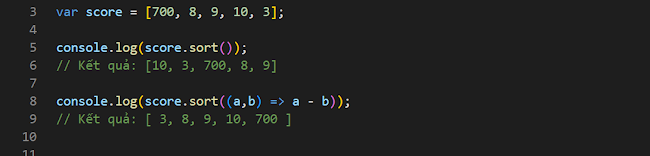
* **total** là biến tích lũy, được trả về sau mỗi lần gọi hàm callback
* **currentValue** là phần tử đang được xử lý
* **currentIndex** là chỉ số của phần tử trong mảng đang được xử lý
* **arr** là mảng hiện tại gọi hàm reduce()
* **initialValue** là giá trị cho tham số thứ nhất. Nếu giá trị này không được cung cấp thì giá trị phần tử đầu tiên của mảng sẽ được sử dụng

###### Sort

* Sort() dùng để sắp xếp các phần tử trong mảng tăng dần hoặc giảm dần theo thứ tự trong bảng mã ascii, hoặc theo quy tắc trong callback function



* Kết quả trên của hàm sort() trả về sai do khi không truyền tham số thì hàm sort sẽ sắp xếp theo ascii.



* Để khắc phục ta sử dụng callback function
* Cú pháp:

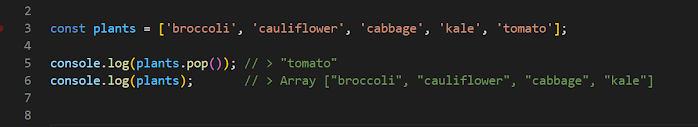
array.sort()

array.sort(function compareFn(firstEl, secondEl) { … })

* Nếu callback trả về số lớn hơn 0 thì secondEl sẽ đứng trước firstEl.
* Nếu trả về số nhỏ hơn hoặc bằng 0 thì giữ nguyên vị trí

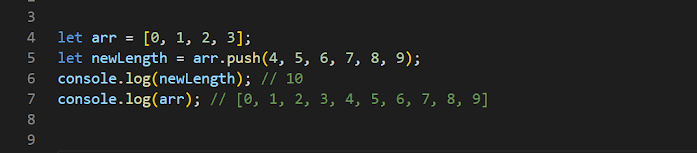
###### Pop

* Pop() dùng để xóa phần tử cuối cùng ra khỏi mảng
* Pop() làm thay đổi mảng ban đầu
* pop() trả về phần tử đã bị xóa ra khỏi mảng
* pop() trả về undefined nếu mảng rỗng



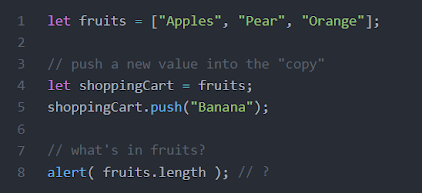
###### Push

* Hàm push() có chức năng thêm phần tử vào cuối mảng
* push() chấp nhận số lượng đối số không giới hạn
* push() trả về độ dài mới của mảng



###### \* Bài tập

1. Đoạn code sau sẽ hiển thị gì



1. Thực hiện các bước làm sau đây
2. Tạo mảng có tên là styles với các mục “Jazz” và “Blues”.
3. Thêm “Rock-n-Roll” vào cuối.
4. Thay thế giá trị ở giữa bằng “Classics”. Code để tìm giá trị giữa sẽ hoạt động với bất kỳ mảng nào có độ dài lẻ.
5. Tách giá trị đầu tiên của mảng và hiển thị nó.
6. Thêm trước Rap và Reggae vào mảng.

3. Viết một hàm filterRange(arr, a, b) lấy một mảng arr, tìm kiếm các phần tử giữa a và b trong đó và trả về một mảng trong số đó.

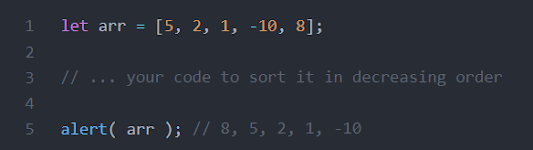
* Không sửa đổi mảng. Trả về mảng mới.

4. Viết một hàm filterRangeInPlace(arr, a, b) lấy một mảng arr và loại bỏ khỏi nó tất cả các giá trị ngoại trừ những giá trị nằm giữa a và b. Nghiệm là: a ≤ arr [i] ≤ b.

Hàm chỉ nên sửa đổi mảng. Nó sẽ không trả lại bất cứ điều gì. Ví dụ:



5. Sắp xếp mảng giảm dần…như ví dụ dưới



6. Chúng ta có một mảng chuỗi arr. Chúng ta muốn có một bản sao được sắp xếp của nó, nhưng giữ cho bản sao không bị sửa đổi.

* Tạo một hàm copySorted(arr) trả về một bản sao như vậy.

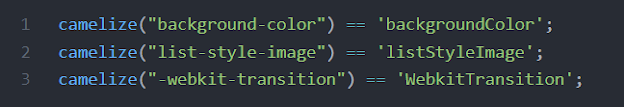


7\*. Đầu vào là một mảng số, ví dụ: arr = [1, -2, 3, 4, -9, 6].

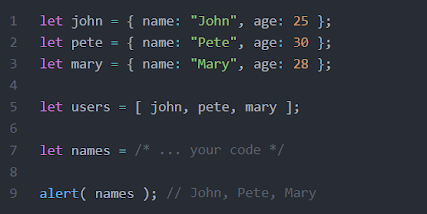
* Nhiệm vụ là: tìm mảng con liền kề của arr với tổng các item lớn nhất.
* Viết hàm getMaxSubSum(arr) sẽ trả về tổng đó.

8. Viết hàm camelize(str) để thay đổi các từ được phân tách bằng dấu gạch ngang như “my-short-string” thành “myShortString” được viết bằng camel.

Đó là: loại bỏ tất cả các dấu gạch ngang, mỗi từ sau dấu gạch ngang trở thành chữ hoa. Ví dụ:



9. Bạn có một mảng các đối tượng người dùng, mỗi đối tượng có user.name. Viết code chuyển đổi nó thành một mảng tên.



10. ạn có một mảng các đối tượng người dùng, mỗi đối tượng có name, surname và id.

* Viết code để tạo một mảng khác từ nó, gồm các đối tượng có id và fullName, trong đó fullName được tạo từ name và surname.

11. Viết hàm sortByAge(users) lấy một mảng các đối tượng User(có name và age)có thuộc tính age và sắp xếp chúng theo độ tuổi.

12. Viết hàm getAverageAge(users) lấy một mảng đối tượng có thuộc tính tuổi và trả về tuổi trung bình.

13. Viết chương trình kiểm tra 1 số có là số hoàn hảo không? Số hoàn hảo là số nguyên dương có tổng các ước dương bằng 2 lần nó. VD số 6 có các ước 1, 2, 3, 6 và tổng 1 + 2 + 3 + 6 = 12 (bằng 2 lần 6)

14. Cho một mảng n số nguyên. Đếm xem trong mảng mỗi số xuất hiện bao nhiêu lần trong mảng. Các số trong mảng có độ lớn từ 0 đến 1000. n <= 100. Yêu cầu:

Đầu vào: [1, 2, 8, 4, 1, 3, 1, 8, 3]

Đầu ra:

[

{ 1 : 3 },

{ 2 : 1 },

{ 3 : 2 },

{ 4 : 1 },

{ 8 : 2 }

]

15. Cho một mảng n số nguyên. Tìm số xuất hiện nhiều nhất trong mảng. Các số trong mảng có độ lớn từ 0 đến 1000. n <= 100

16. Viết chương trình kiểm tra năm nhuận

17. Viết chương trình chuyển đổi số từ hệ thập phân sang nhị phân và ngược lại

18. Viết chương trình tính số lượng chữ số của một số tự nhiên n

( n < )

19. Cho 1 mảng số, tìm số cặp số bằng nhau liên tiếp

20. Nhập vào mảng gồm n số nguyên. Tìm các thứ tự (vị trí) của số lớn nhất trong mảng.

21. Cho mảng sau

[

{ id : 15 }, { id : - 1 }, { id : 0 }, { id : 3 },

{ id : '12' }, { id : null }, { id : NaN }, { id : 'undefined' }

]

Lọc ra các giá trị id là số dương trong mỗi object.

22. Viết chương trình để tính cước điện thoại bàn cho một hộ gia đình với các thông số như sau:

* Phí thuê bao bắt buộc là 25 nghìn.
* 600 đồng cho mỗi phút gọi của 50 phút đầu tiên.
* 400 đồng cho mỗi phút gọi của 150 phút tiếp theo.
* 200 đồng cho bất kỳ mỗi phút nào sau 200 phút đầu tiên.

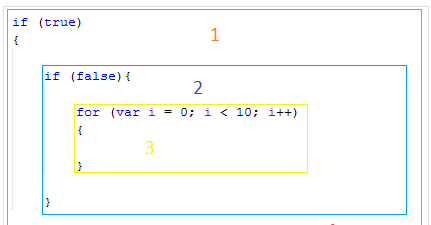
23. Nhập vào một số nguyên n ( 0 < n < 1 triệu) in ra số đảo ngược của số n.

##### Oop

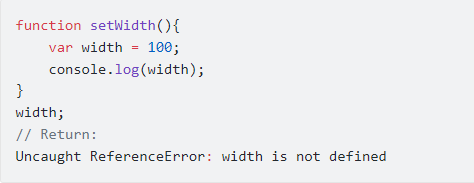
##### Es6

###### Block scope

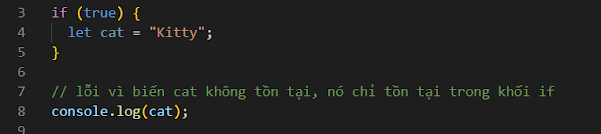
* Block Scoped là phạm vi trong một khối, nghĩa là chỉ hoạt động trong phạm vi được khai báo bời cặp {}.



* **Var:** là function scope. Nó chỉ có tác dụng trong phạm vi function được khai báo. Vậy nếu khai báo bên ngoài function thì nó là biến toàn cục. Nếu khai báo 1 biến var bên trong function và gọi biến đó bên ngoài function thì biến đó không hoạt động ở phạm vi ngoài function Ví dụ

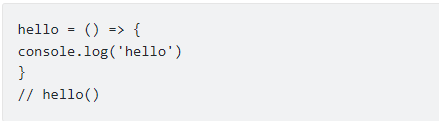


* **const, let** là block-scope. const và let dùng để khởi tạo biến nhưng biến đó chỉ có tác dụng bên trong khói đang khảo báo. biến được khai báo bằng const là hằng sẽ không thay đổi được giá trị
* VD:

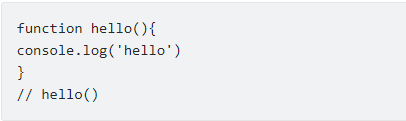


###### Arrow function

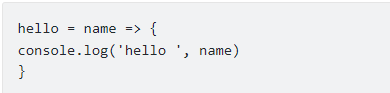
* Arrow Function (hay còn gọi là hàm mũi tên) là một trong những tính năng mới và rất hay của ES6. Nó thừa hưởng cách viết ngắn gọn bằng cú pháp ES6. Tuy Arrow Function và Function đều là function nhưng cách viết và cách tham chiếu tới bối cảnh (context) khác nhau.
* Sự khác nhau
* Arrow function sử dụng kí tự =>



* Function thông thường



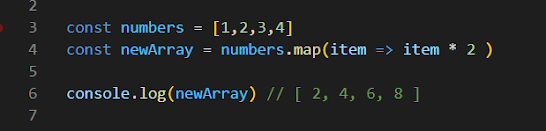
* Trường hợp có tham số



* Ta có thể giản lược dấu () khi chỉ có 1 tham số truyền vào
* Arrow function ta có thể bỏ qua return

b

* Ta có thể kết hợp array function trong trường hợp dùng map, filter, forEach



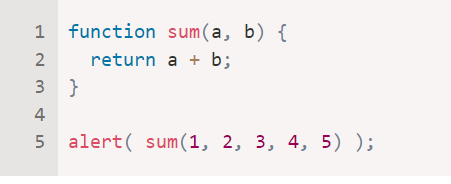
+ Arrow function không có bind, nên biến this chỉ đến context gần nhất



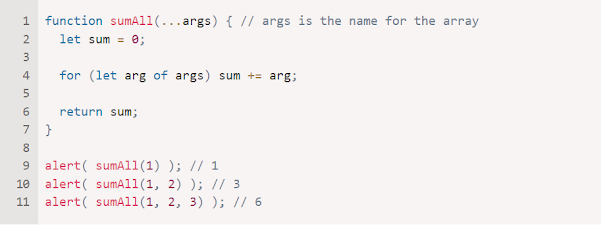
* Bài tập: viết hàm giải phương trình bậc 2 bằng arrow function

###### Rest parameters

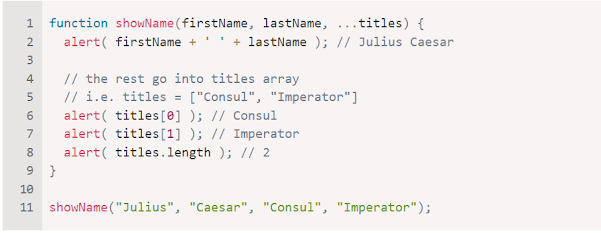
* Một function có thể được gọi với số lượng đối số bất kì:



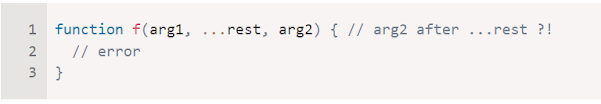
* Sẽ không xảy ra lỗi nếu truyền quá nhiều đối số vào function. Và kết quả sẽ chỉ tính 2 giá trị đầu tiên
* Phần còn lại của các tham số có thể được bao gồm bên trong phần định nghĩa function bằng cách sử dụng 3 dấu chấm … tiếp sau đó là tên tên của mảng sẽ chứa các đối số.



* Ta có thể chọn các đối số đầu tiên và thêm các phần còn lại vào phía sau

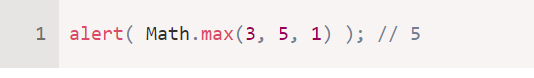


\*\*\* Lưu ý: Các đối số được thêm vào sau bắt buộc phải ở cuối cùng

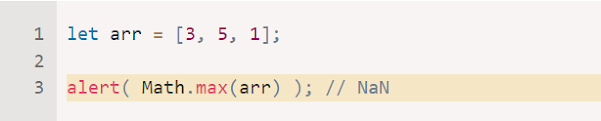


###### Spread syntax

* Phía trên ta đã biết cách chuyển danh sách đối số thành 1 mảng
* Nhưng đôi khi ta phải làm ngược lại
* Ví dụ: Hàm max sẽ truyền vào danh sách các số và trả về kết quả



* Còn khi truyền vào 1 mảng số thì sẽ bị lỗi



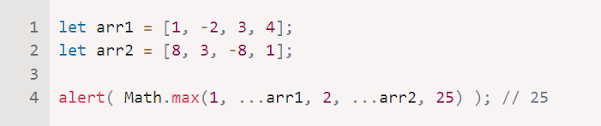
* Và *Spread syntax* sẽ xử lý được điều đó.
* Khi sử dụng … arr khi gọi function, nó sẽ “mở rộng” đối tượng mảng thành danh sách các đối số



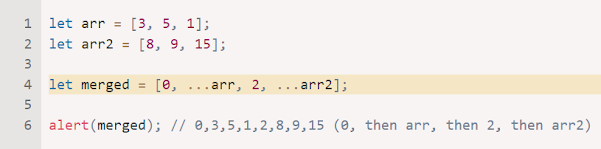
* Ta có thể truyền nhiều mảng



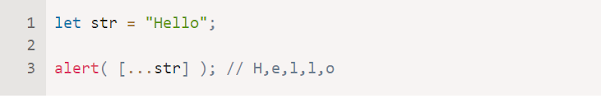
* Có thể kết hợp với các giá trị bình thường



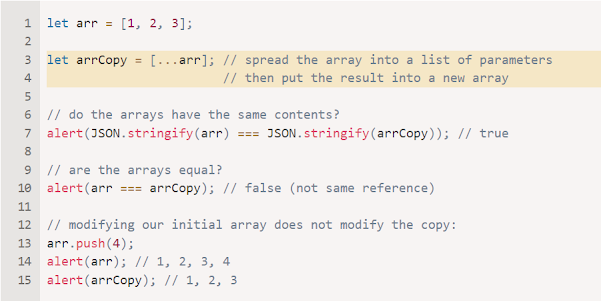
* Ngoài ra, phương pháp này ta có thể kết hợp các mảng với nhau

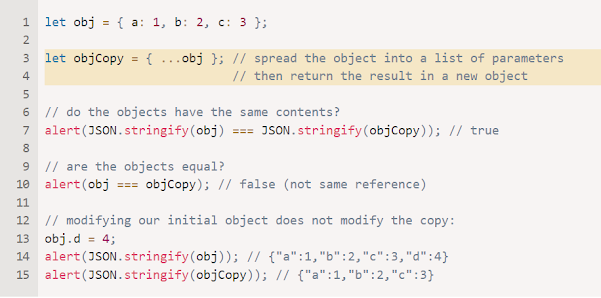


* Ta có thể tách 1 mảng thành từng chuỗi ký tự



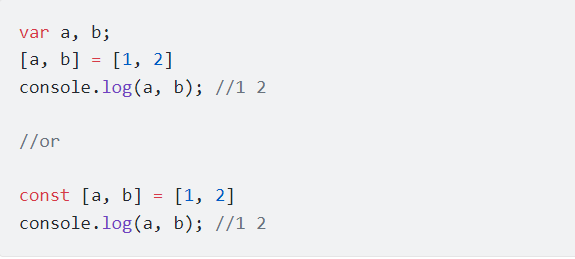
* Cú pháp này giống như ta sử dụng for … of
* Sao chép array/object
* Ta có thể sao chép 1 mảng hay 1 object bằng spread syntax.



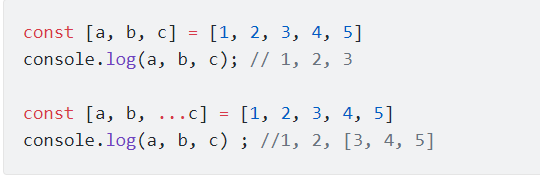


###### Destructuring

* **Destructuring** là một cú pháp cho phép bạn gán các thuộc tính của một Object hoặc một Array. Điều này có thể làm giảm đáng kể các dòng mã cần thiết để thao tác dữ liệu trong các cấu trúc này. Có hai loại Destructuring: Destructuring Objects và Destructuring Arrays.
* Ví dụ 1:



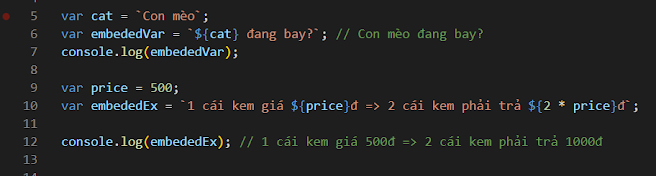
* Trong dòng 1, chúng ta có 2 biến a và b trong array. Trong dòng tiếp theo, chúng ta thiết lập chúng và set chúng bằng với một arrays tương ứng. Trong các dòng tiếp theo, ta in các giá trị của a & b và nhận được 1 và 2 tương ứng là các phần tử trong mảng phía bên phải. Đó chính là Destructuring javascript.
* Ví dụ 2:



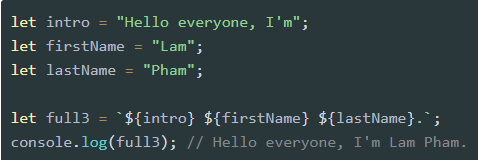
* Phía trên thì các giá trị a, b, c tương ứng là 3 giá trị đầu tiên của array bên phải. Còn phía dưới với rest params (... c ) thì nó sẽ nhận các giá trị còn lại của mảng

###### Template string

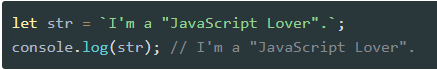
* Template Literals hay còn gọi là Template Strings là một cú pháp mới để khai báo String trong Javascript
* Cú pháp



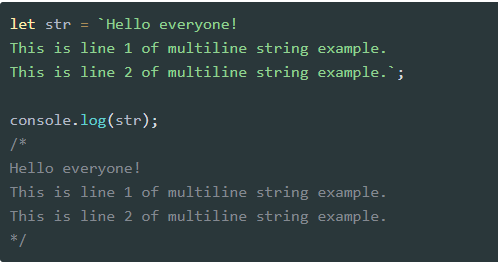
* Các ưu điểm
* Ghép nối string



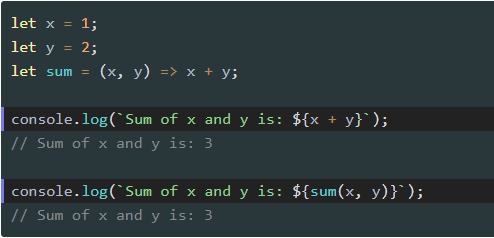
* Sử dụng đồng thời nhảy đơn, nhảy kép



* Viết string trên nhiều dòng



* Viết biểu thức toán học hoặc function ngay trong string

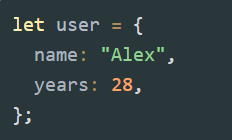


* Viết HTML template



Bài tập:

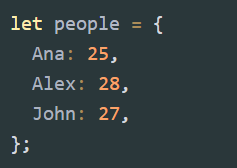
1. Cho object:



Sử dụng destructuring assignment trong JavaScript thực hiện các yêu cầu sau:

1. Đọc giá trị thuộc tính name và gán vào biến name.
2. Đọc giá trị thuộc tính years và gán vào biến age.
3. Đọc giá trị thuộc tính isAdmin và gán vào biến isAdmin (nếu thuộc tính không tồn tại thì giá trị mặc định là false).

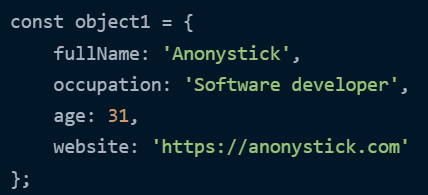
2.Cho object:



* Viết hàm topAge(people) trả về tên của người nhiều tuổi nhất.

1. Nếu people là rỗng thì hàm trên trả về null.
2. Nếu có nhiều người có cùng tuổi nhiều nhất thì trả về một người bất kỳ.

3. Cho object



1. In ra tất cả thông tin trên cùng 1 dòng
2. In ra tất cả thông tin trên từng dòng
3. Tạo 1 object 2 với thông tin giống object phía trên
4. Tạo 1 object thứ 3 và thay đổi fullname là “Tom”, age là “25”.

4. Viết arrow function đổi giá trị của 2 số với nhau

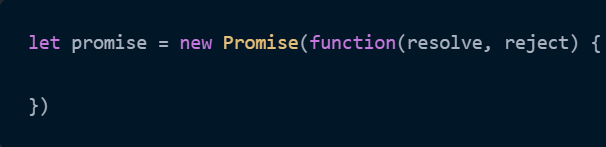
5. Cho object



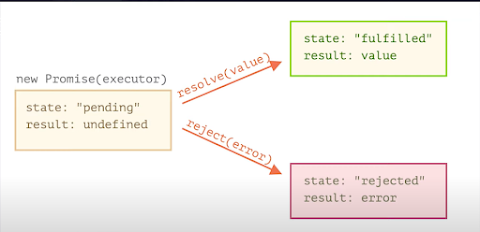
* Gán các biến width, height, item1, item2 tương ứng với các giá trị trên và in ra màn hình.

###### Promise

* Nhiều trường hợp bắt buộc trong lập trình chúng ta phải lập trình đồng bộ, nghĩa là bạn phải chạy function 1 để có data 1, dùng data 1 để query data 2. Mà trong javascript là chạy bất đồng bộ, nên chính lúc này bạn phải sử dụng Promise
* Cú pháp:



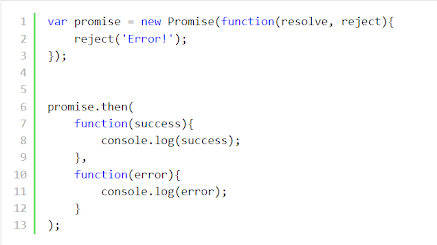
* Khi promise thực thi hoàn thành công việc, nó sẽ gọi một trong các hàm mà nó nhận làm đối số. Đó là resolve(value) và reject(error)



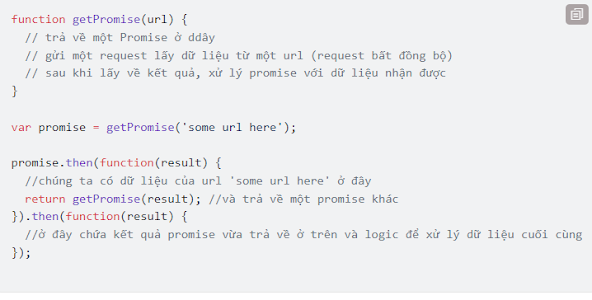
* Mỗi đối tượng Promise sẽ có một thuộc tính private chứa trạng thái hiện tại của Promise.
* Ví dụ resolve



* Ví dụ reject



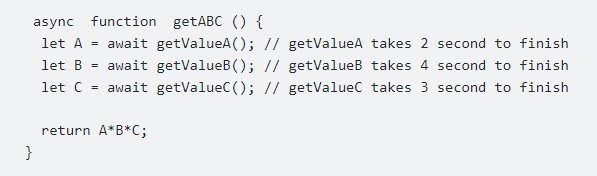
* Xử lý callback hell



###### Async - await

1. Async - await là gì?

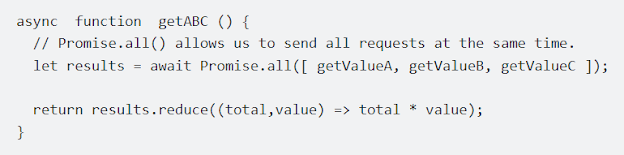
* Async / Await là một tính năng của JavaScript giúp chúng ta làm việc với các hàm bất đồng bộ theo cách thú vị hơn và dễ hiểu hơn. Nó được xây dựng trên Promises
* **Async** - khai báo một hàm bất đồng bộ
  + Tự động biến đổi một hàm thông thường thành một Promise.
  + Khi gọi tới hàm async nó sẽ xử lý mọi thứ và được trả về kết quả trong hàm của nó.
  + Async cho phép sử dụng Await.
* **Await** - tạm dừng việc thực hiện các hàm async.
  + Khi được đặt trước một Promise, nó sẽ đợi cho đến khi Promise kết thúc và trả về kết quả.
  + Await chỉ làm việc với Promises, nó không hoạt động với callbacks.
  + Await chỉ có thể được sử dụng bên trong các function async.
* Ví dụ



* Mỗi lần gọi tới hàm await sẽ đợi cho đến khi hàm await trước đó kết thúc. Vì các wait sẽ đợi và thực hiện tuần tự từng cái một, toàn bộ chức năng sẽ mất 9 giây để thực hiện xong hàm từ đầu đến cuối (2 + 4 + 3).
* Async - await là cách viết gọn hơn cho promise.

###### Vậy khi có Async / Await có làm cho promises lỗi thời?

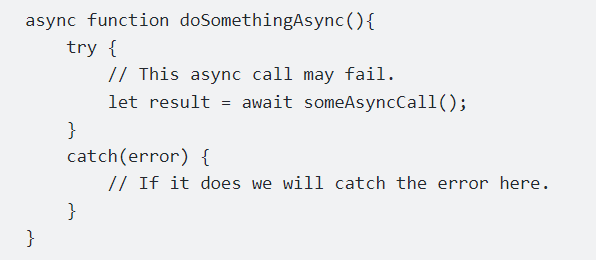
* Như ví dụ ở trên, function getABC mất 9s để có thể trả ra kết quả
* Đây không phải là một giải pháp tối ưu vì A, B và C không phụ thuộc vào nhau, chúng ta không cần biết giá trị của A trước khi chúng ta có được B. Vì vậy, chúng ta có thể lấy chúng cùng một lúc và thời gian chờ đợi sẽ được giảm bớt đi.
* Trong trường hợp như thế này, sử dụng Promise sẽ thích hợp hơn. Để gửi tất cả các yêu cầu cùng lúc, chúng ta sử dụng Promise.all(). Việc sử dụng Promise.all() sẽ đảm bảo chúng ta có tất cả các kết quả trước khi tiếp tục thực thi code nhưng việc gọi đến các hàm bất đồng bộ sẽ được chạy song song mà không phải tuần tự từng cái một



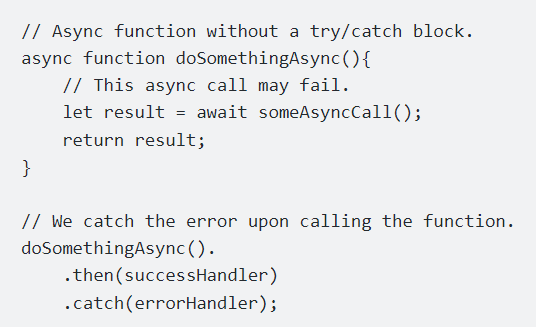
* Bằng cách này thời gian thực thi hàm sẽ nhanh hơn. 3 hàm sẽ thực hiện song song, chúng ta sẽ chỉ mất 4s để chờ cả 3 hàm trả về giá trị

1. Xử lý lỗi

* Xử dụng try catch



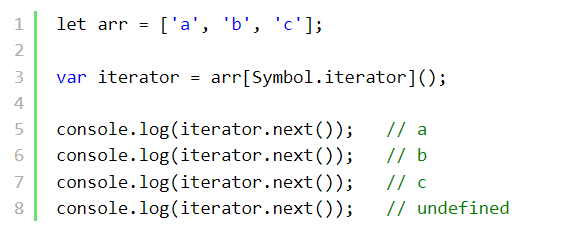
* Vì async cũng là 1 promise nên ta có thể sử dụng then và catch



###### Iterator - iterable

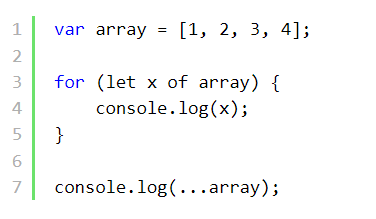
1. Iterator

* Iterators là một bộ duyệt dùng để duyệt qua một mảng, một danh sách hoặc một collection mà qua mỗi lần duyệt sẽ ghi lại vị trí đã duyệt để từ đó có thể biết và lấy vị trí tiếp theo.
* Iterators có cung cấp phương thức next() và phương thức này sẽ return về phần tử kết tiếp, đồng thời ghi nhận luôn phần tử đã lặp là phần tử next(). Phương thức next() sẽ return về một Object gồm hai thuộc tính là value và done.



1. Iterable

* Iterable là khả năng cho phép các đối tượng trong Javascript sử dụng các kỹ thuật xử lý dữ liệu như for of loop, toán tử ba chấm …



* Iterator Protocol là các giao thức (*method*) xử lý một đối tượng **có đánh dấu vị trí đã duyệt**, vì vậy với các đối tượng thông thường sẽ không sử dụng được nên ta phải sử dụng Symbol.iterator để chuyển đổi.

### Express

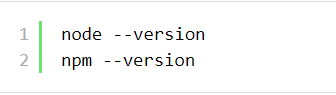
#### Expressjs là gì?

* **Expressjs** là một framework được xây dựng trên nền tảng của **Nodejs**. Nó cung cấp các tính năng mạnh mẽ để phát triển web hoặc mobile. **Expressjs** hỗ trợ các method HTTP và middleware tạo ra API vô cùng mạnh mẽ và dễ sử dụng.

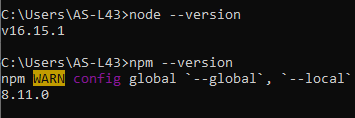
#### Setup

B1: Kiểm tra Nodejs và npm được cài đặt hay chưa

* Mở cmd dùng 2 lệnh sau

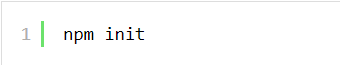


* Nếu nhận được 2 dòng sau thì đã cài thành công



B2: Dùng npm cài đặt Expressjs

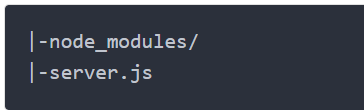
* Tạo 1 file lưu dự án
* Mở CMD
* Chạy lệnh sau



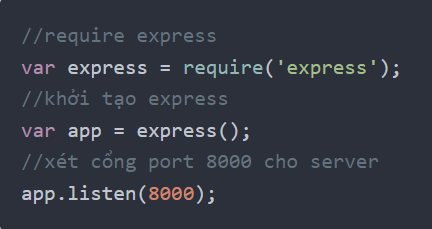


#### Chạy chương trình đầu tiên

B1: Tạo 1 file server.js nằm ở local như sau



* File server.js có nội dung sau



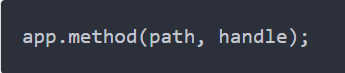
B2: Thêm routing



B3: Chạy lệnh node server.js trên terminal

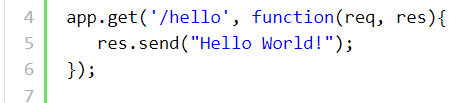
#### Routing trong express

* Router trong express framework là một bộ định tuyến giúp cho chúng ta định danh ra các url và hành động kèm theo nó
* Cú pháp

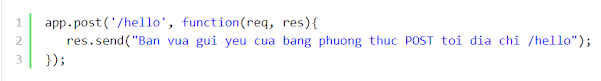


Trong đó:

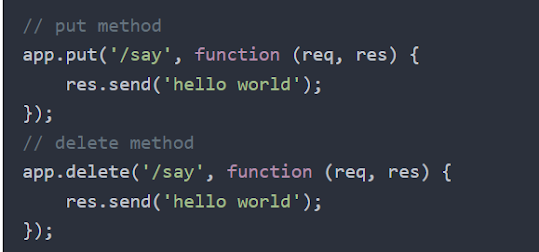
* app là biến mà khi chúng ta khởi tạo express framework ([xem bài trước](https://toidicode.com/chay-truong-trinh-dau-tien-voi-express-framework-190.html)).
* method là một trong các dạng http method sau: get,post,put,delete,head,path
* path là thành phần phía sau domain mà chúng ta muốn xác định.
* handle là hành động sẽ thực thi khi danh path được gọi. handle ở đây là callback function Với req là biến chứa tất cả các thông số request mà người dùng gửi lên và res là biến chứa tất cả các thông số mà server trả về cho client.
* Các ví dụ
* Router GET method: route chỉ nhận phương thức get đến url, phương thức này thường dùng để hiển thị dữ liệu



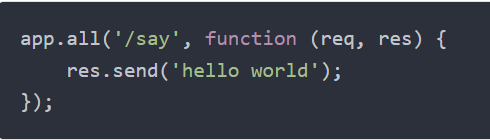
* Router POST method: route chỉ nhận phương thức post, Phương thức này thường dùng để gửi dữ liệu lên cho server



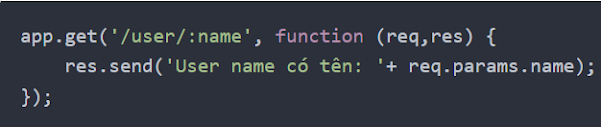
* Các router khác



* Nhận tất các các phương thức



* Truyền tham số vào router



#### \* Bài tập

* Xây dựng 1 chương trình to do list. Với các chức năng CRUD. Dữ liệu lưu bằng mảng local.

## Section 2:

### Typescript

* Định nghĩa: TypeScript là một dự án mã nguồn mở được phát triển bởi Microsoft, nó có thể được coi là một phiên bản nâng cao của Javascript bởi việc bổ sung tùy chọn kiểu tĩnh và lớp hướng đối tượng mà điều này không có ở Javascript.

### Setup

#### Cài đặt môi trường

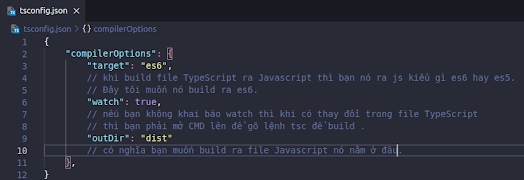
* run command: npm install -g typescript
* check version: tsc –version

#### Chạy code

* TypeScript được viết trong các file **.ts** (hoặc **.tsx** cho **JSX**), nó không thể sử dụng trực tiếp trong trình duyệt và cần biên dịch thành JavaScript.
* Câu lệnh sau sẽ nhận một file TypeScript là *main.ts* và chuyển nó thành JavaScript *main.js*. Nếu *main.js* đã tồn tại nó sẽ bị ghi đè: tsc \*\*\*.ts

#### Tsconfig.json

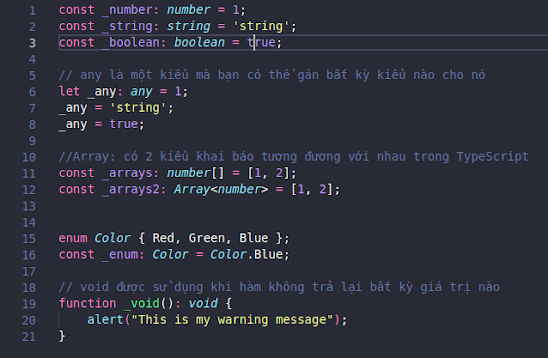
* Là file để xây dựng môi trường build, các tùy chọn để biên dịch typescript.
* Một số tùy chọn trong file tsconfig.json:



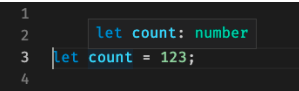
### Type

#### Basics type

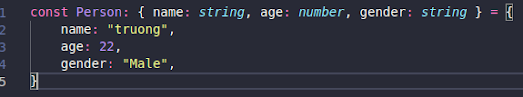
* Trong TypeScript chia ra làm 7 loại cơ bản, bao gồm: boolean, number, string, array, enum, any, void. khi khai báo ta sẽ sử dụng cấu trúc như sau: tên\_biến : kiểu\_trả\_về = giá\_trị\_biến;



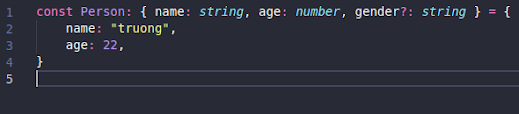
* Inferred type: Typescript thông minh detect được kiểu dữ liệu tương ứng ngay cả khi mình không khai báo cụ thể kiểu dữ liệu.



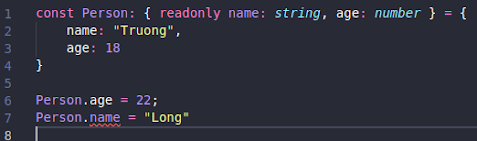
* Object type:
* Ngoài các kiểu dữ liệu primitives, kiểu dữ liệu phổ biến nhất mà bạn gặp là Object type. Để xác định một kiểu đối tượng chúng ta chỉ cần liệt kê các thuộc tính của nó và các kiểu của chúng.



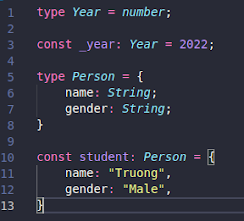
* Optional Properties: Ngoài ra Object type cũng có thể chỉ định một số hoặc tất cả các thuộc tính của chúng là tùy chọn bằng cách thêm dấu ? sau tên thuộc tính:



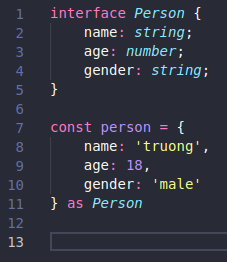
* ReadOnly Properties: khi được gán readonly, thuộc tính đó sẽ chỉ được khởi tạo mà không được thay đổi giá trị.



* Type aliases: Với type aliases, chúng ta có thể tạo tên mới cho một type nhưng chúng ta không define một type mới

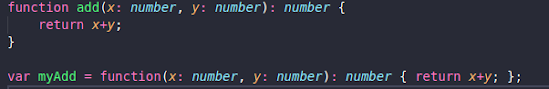


* Type Assertions: Khi sử dụng Type Assertions, trình biên dịch sẽ coi một giá trị là một kiểu được chỉ định cụ thể. Nó sử dụng từ khóa as để sử dụng

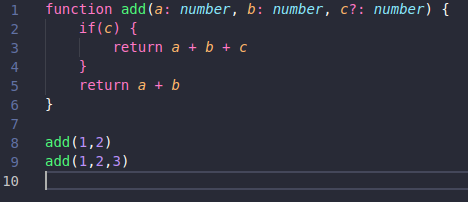


#### Function

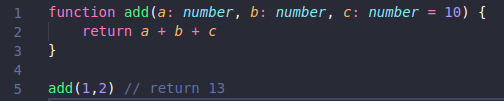
* Cũng giống như javaScript, typeScript có 2 cách khai báo function. Nhưng khi khai báo function trong typeScript còn hỗ trợ việc khai báo với các kiểu trả ra của function và cũng như kiểu đầu vào của dữ liệu

****

* Optional Parameters: trong javascript, mỗi hàm sẽ cố định số lượng tham số, nhưng trong typescript có hỗ trợ option parameters giúp một hàm có thể linh hoạt hơn rất nhiều.



* Default parameters: Trong cú pháp này, nếu bạn không truyền các đối số hoặc truyền undefined vào bên trong hàm khi gọi nó, hàm sẽ nhận các giá trị khởi tạo mặc định cho các tham số bị bỏ qua.



* Không thể sử dụng chung Optional parameters với default parameters cho cùng một đối số.

#### Union Types

* Đôi khi một biến có thể có nhiều kiểu dữ liệu, thay vì chúng ta định nghĩa từng biến cho mỗi kiểu dữ liệu thì typescript có hỗ trợ union types giúp code trở nên ngắn gọn hơn rất nhiều.



#### Interfaces

* Interface trong typescript cho phép bạn định nghĩa thuộc tính là gì và phương thức là gì mà đối tượng cần để được implement. Nếu đối tượng tuân thủ đúng khuôn mẫu interface thì đối tượng đã implement interface ấy sẽ được thực thi đúng. Nếu interface không được thực thi đúng thì typescript sẽ phát sinh lỗi ngay lập tức.

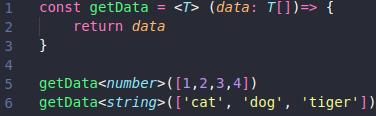


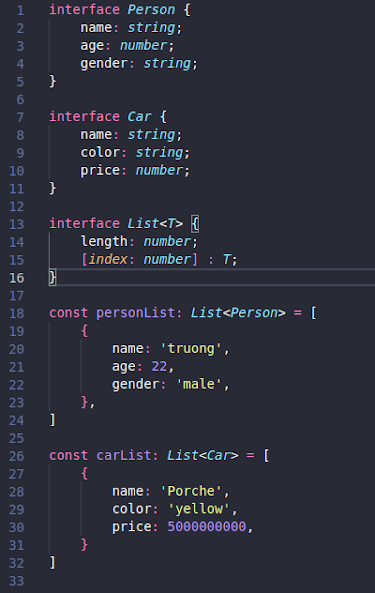
* Sự khác nhau giữa Interface và Type

|  |  |
| --- | --- |
| Interface | Type |
| Có thể khai báo lại | Không thể khai báo lại |
| Không sử dụng được với unions | Có thể sử dụng |
| sử dụng extend hoặc khai báo lại để gộp thuộc tính | sử dụng intersection để gộp |

#### Generics

* Generic type trong TypeScript cho phép bạn viết các function, class và interface có thể tái sử dụng và tổng quát hóa. Generic type cho phép bạn nhận tham số và trả về kiểu dữ liệu tương ứng.





#### Một số Utility types phổ biến

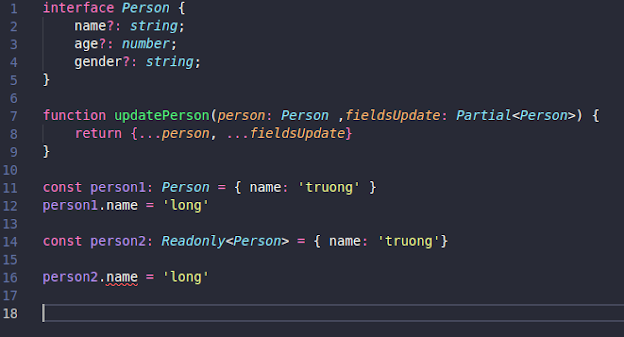
* Partial: Set tất cả thuộc tính của Type thành optional



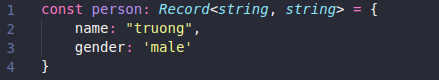
* Required: Set tất cả thuộc tính của Type thành required



* Readonly: Set tất cả thuộc tính của Type thành readonly



* Record<Keys, Type>: Định nghĩa của kiểu type với key và value



### Express

#### Cơ bản:

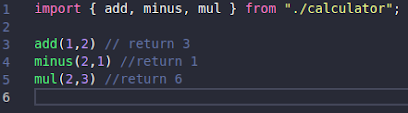
#### Deep down:

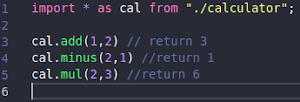
##### Module import

* Module là những đoạn mã được đóng gói lại và được giữ Private. Điều đó có nghĩa là các hàm hay biến trong module mới có thể truy cập và thao tác với nhau. Việc đóng gói các đoạn code thành các module làm cho việc phát triển và bảo trì hệ thống được dễ dàng, làm giảm sự phục thuộc các phần trong chương trình. Module còn có tính tái sử dụng cao giúp cho việc code trở nên sạch sẽ hơn.
* Để sử dụng được module đó, chúng ta cần import chúng vào trong file cần sử dụng.
* Export: Sử dụng để export một module ra bên ngoài, nếu không sử dụng export thì module sẽ ở dạng private và không sử dụng được ở bên ngoài.

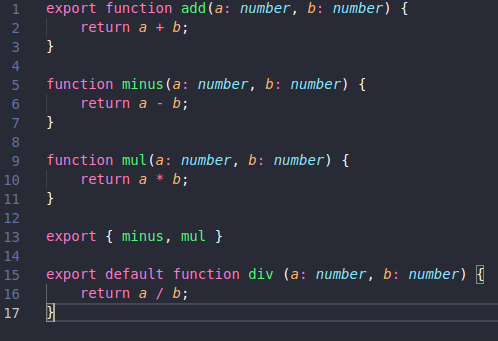


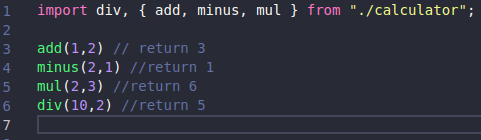
* Import: Sử dụng để import một module vào file cần sử dụng





* Default Export: Default Export trong Javascript ES6 chỉ cho phép xuất một mặc định cho mỗi file. Default Export có thể cho một function, class hoặc một object. Default cho phép bạn đặt lại tên cho module.





##### Package json

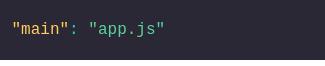
* 1. Tệp package.json là phần cốt lõi của hệ sinh thái Node.js và là phần cơ bản nhất để hiểu và làm việc với Node.js, npm và cả JavaScript. Nó được sử dụng như một bản kê khai thông tin ứng dụng, các modules, thư viện và nhiều thứ khác được sử dụng trong ứng dụng của chúng ta.
  2. Identifying metadata
* Thuộc tính name: là thành phần cơ bản của package.json. name là một chuỗi xác định tên của ứng dụng



* Thuộc tính version: là một phần quan trọng của package.json, vì nó biểu thị bản hiện tại của module mà package.json đang mô tả

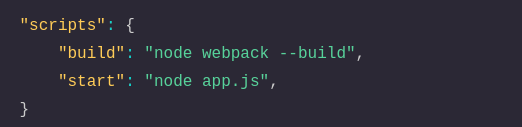


* Thuộc tính main: sẽ chỉ ra phần xử lý chính của dự án, module mà bạn phát triển. Trong một ứng dụng nodejs, mặc định khi chúng ta khởi tạo nó sẽ gọi đến một file có dạng file.js. Ở đây có nghĩa là nó sẽ lấy file.js làm phần gọi để xử lý chính của dự án, module của bạn.



* Thuộc tính repository: được định nghĩa là một object bên trong package.json. Thông thường nó sẽ chứa type và url vè kho lưu trữ github.



* Thuộc tính script: là các câu lệnh thực thi bên trong dự án. 
* Thuộc tính dependencies: là nơi chứa các name và version của một thư viện, một dự án nào đó hỗ trợ cho ứng dụng mà bạn đang xây dựng. Các thư viện này sẽ bị ràng buộc bởi version của nó cho dự án mãi mãi cho đến khi bạn thay đổi hoặc upgrade nó.



* Thuộc tính devDependencies: gần giống với thuộc tính dependencies về mặt cấu trúc, với một điểm khác biệt chính. Thuộc tính dependencies được sử dụng để xác định các thư viện phụ thuộc mà một ứng dụng cần chạy khi đấy lên phiên bản product. Thuộc tính devDependencies thường được sử dụng để xác định các phụ thuộc mà module cần để chạy trong quá trình dev.



##### Npm, npx, nvm

###### NPM

* Npm (node package manager) là một công cụ quản lý các thư viện chúng ta có được khi cài đặt nodejs (npm được cài sẵn khi chúng ta cài nodejs). Nó cung cấp cho lập trình viên cách để cài đặt những thư viện sử dụng ở toàn bộ dự án của mình hoặc có thể gói gọn trong một dự án.
* Npm bao gồm 3 thành phần:
* Website: hiển thị một cách chi tiết nhất những thư viện được chia sẻ
* Command line Interface (CLI): công cụ quản lý để lấy những thư viện về
* Registry: lưu trữ những thư viện javascript cũng như thông tin về những thư viện đó.

###### NPX

* Npx (npm package executor): khi bạn muốn xem xét một số thư viện và thực thi một số lệnh, nhưng bạn không muốn cài vào nội bộ dự án của mình, thì bạn có thể sử dụng npx



###### NVM

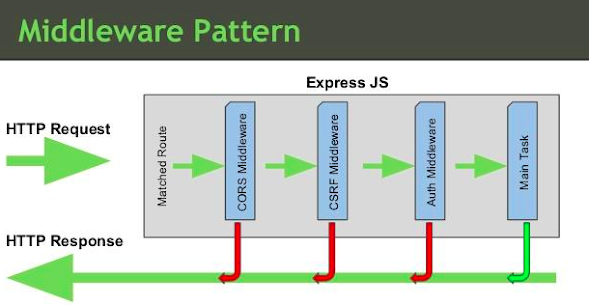
* Nvm (node version manager) là một package tool được phát triển trên github để quản lý phiên bản node. Lý do ta sử dụng nvm là khi gặp tình trạng quản lý và làm việc với nhiều project có các thời gian phát hành khác nhau, nên không tương thích với node mới nhất. Ta sẽ sử dụng nvm để cài đặt các phiên bản node.

nvm install 8.9.1

##### Middleware

###### Định nghĩa

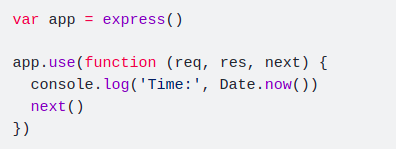
* Là các hàm được dùng để tiền xử lý, lọc các request trước khi đưa vào xử lý logic hoặc điều chỉnh các response trước khi gửi về cho người dùng.



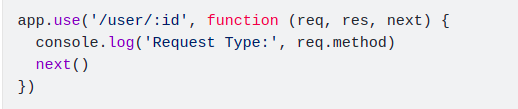
* Hình trên mô tả 3 middleware có trong ExpressJS. Một request khi gửi đến Express sẽ được xử lý qua 5 bước như sau:
* Tìm Route tương ứng với request
* Dùng CORS Middleware để kiểm tra cross-origin Resource sharing của request
* Dùng CSRF Middleware để xác thực CSRF của request, chống fake request
* Dùng Auth Middleware để xác thực request có được truy cập hay không
* Xử lý công việc được yêu cầu bởi request
* Bất kỳ bước nào trong bước 2, 3, 4 nếu xảy ra lỗi sẽ trả về response thông báo cho người dùng, có thể là lỗi CORS, lỗi CSRF, hay lỗi auth tùy thuộc vào request bị dừng ở bước nào.
* Các chức năng mà middleware có thể thực hiện trong ExpressJS sẽ bao gồm:
* Thực hiện bất cứ đoạn code nào
* Thay đổi các đối tượng request và response
* Kết thúc một quá trình request-response
* Gọi hàm middleware tiếp theo trong stack
* Trong Express có 5 kiểu middleware có thể sử dụng
* Application-level middleware (middleware cấp ứng dụng)
* Router-level middleware (middleware cấp điều hướng - router)
* Error-handling middleware (middleware xử lý lỗi)
* Built-in middleware (middleware sẵn có)
* Third-party middleware (middleware của bên thứ ba)

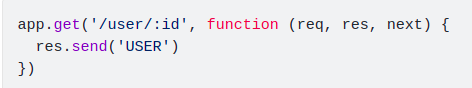
###### Application-level middleware

* Khi khởi tạo một Web Application với Express, chúng ta sẽ có một đối tượng đại diện cho Web App đó, thường được gán với biến app. Đối tượng này có thể khai báo các middleware thông qua các hàm: app.use() hoặc app.METHOD (method sẽ là các kiểu HTTP Method được Express hỗ trợ như app.get().



* ví dụ trên mô tả một hàm không khai báo đường dẫn cụ thể do đó nó sẽ được thực hiện mỗi lần request.





* Ví dụ trên dùng hàm use đến đường dẫn /user/:id . Hàm này sẽ được thực hiện mỗi khi request đến đường dẫn /user/:id.
* Khi muốn gọi một loạt các middleware cho một đường dẫn cụ thể ta có thể thực hiện như dưới đây:



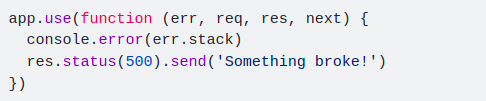
###### Router-level middleware

* Các middleware này về chức năng không khác gì so với các application-level middleware ở trên, tuy nhiên thay vì dùng biến app có thể gây nhầm lẫn với các thiết lập, phần router có thể không rõ ràng và khó phân biệt, Express cung cấp một đối tượng router chuyên dùng để khai báo route bằng cách gọi hàm sau:



###### Error-handling middleware

* Đây là các middleware phục vụ việc xử lý lỗi. Một lưu ý là các hàm cho việc này luôn nhận 4 tham số (err, req, res, next). Khi muốn khai báo một middleware cho việc xử lý lỗi, bạn cần tạo một hàm có 4 tham số đầu vào. Mặc dù bạn có thể không cần sử dụng đối tượng next, nhưng hàm vẫn cần format với bốn tham số như vậy. Nếu không ExpressJS sẽ không thể xác định đó là hàm xử lý lỗi, và sẽ không chạy khi có lỗi xảy ra, chỉ hoạt động giống như các hàm middleware khác.



###### Built-in middleware

* Kể từ phiên bản 4.x, ngoài middleware express.static, express.json, express.urlencoded, tất cả các hàm middleware khác đều đã tách ra thành các module riêng biệt.
* Express.static: dựa trên thư viện serve-static, được dùng để cung cấp các nội dung tĩnh trong trang web, ví dụ như các trang HTML tĩnh, các file hình ảnh, css, js, …Đoạn code dưới đây mô tả việc sử dụng express.static để tạo ra một thư mục có tên public.



* Express.json: là một phần mềm trung gian nhanh được xây dựng giúp chuyển đổi phần body request thành JSON.
* Express.urlencoded: là một phương thức được xây dựng sẵn để nhận request đến dưới dạng chuỗi hoặc mảng.

###### Third-party middleware

* Sử dụng third-party sẽ giúp chúng ta thêm các chức năng cho Web App của mình mà không cần mất nhiều công implement.
* Chúng ta sẽ cần cài đặt module thông qua npm, sau đó khai báo sử dụng trong đối tượng app, nếu dùng ở Application-level, hoặc đối tượng router nếu dùng ở Router-level
* Đoạn code sau sẽ cài đặt và sử dụng một middleware có tên là cookie-parser dùng để đọc cookies của request:



* Một số third-party khác như: body-parser, cors, morgan, …

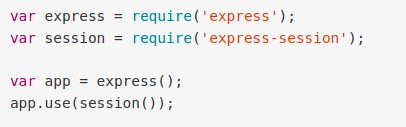
##### Session

###### Định nghĩa

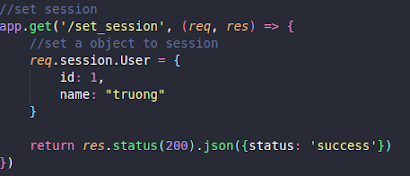
* Session đơn giản là 1 cách để chúng ta lưu lại dữ liệu của người dùng sử dụng website. Giá trị của session được lưu trong một tập tin trên máy chủ. Ví dụ khi bạn đăng nhập vào một trang web và đăng nhập với tài khoản đã đăng ký trước đó. Máy chủ sau khi xác thực được thông tin bạn cung cấp là đúng nó sẽ sinh ra một tập tin (hay chính là session của trình duyệt của bạn) chứa dữ liệu cần lưu trữ của người dùng.

###### Session trong Express

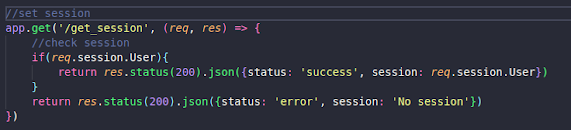
* Setup session trong express



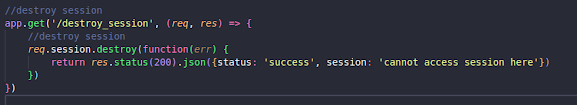
* Store session:
* Cookies: Chúng ta có thể store session trên cookie session nodejs mỗi trình duyệt nhưng chú ý rằng tất cả đều nằm ở Clients.
* Memory cache: Như chúng ta đã biết, Cache được lưu trữ trong bộ nhớ. Chúng ta cũng có thể sử dụng thêm những cache module như Redis và Memcached.
* Set session:



* Get session:



* Delete session:



##### Auth

###### JWT

* JWT được viết tắt từ cụm từ JSON Web Token. Đây là một chuẩn mở (RFC 7519) có thể giúp bạn tạo ra một cái chuỗi mã hóa chứa các dữ liệu để bạn trao đổi thông tin giữa các hệ thống khác nhau một cách an toàn và đáng tin cậy. Các chuỗi thông tin thì sẽ được mã hóa một cách ngẫu nhiên, tùy hứng và không theo một trật tự sắp xếp nào.
* Cấu trúc của JSON Web Token:
* Header: Phần **header** sẽ chứa kiểu dữ liệu , và thuật toán sử dụng để mã hóa ra chuỗi JWT



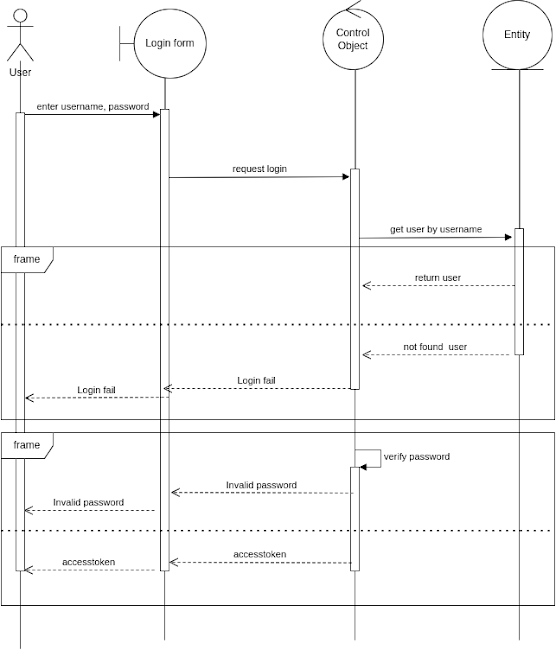
* Payload: Phần payload sẽ chứa các thông tin mình muốn đặt trong chuỗi Token như username, userId, author…



* Signature: Phần chữ ký này sẽ được tạo ra bằng cách mã hóa phần header, payload kèm theo một chuỗi secret.
* JWT thường được dùng trong phần authentication. Khi người dùng đã đăng nhập vào hệ thống thì những request tiếp theo từ phía người dùng sẽ chứa thêm mã jwt. Điều này cho phép người dùng được cấp quyền truy cập vào các url, service, và resource mà mã token cho phép.

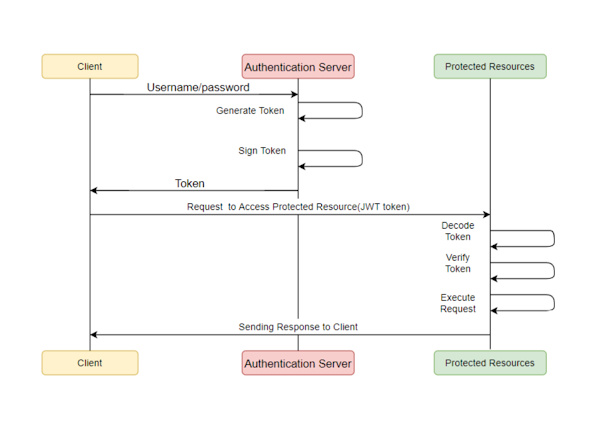
###### Authentication

* Cơ chết authentication xác định xem bạn có phải là user của hệ thống hay không trước khi bạn sử dụng tài nguyên của hệ thống. Trong hầu hết các trường hợp ứng dụng cung cấp một login form với những thông tin nhất định để xác minh người dùng.



###### Authorization

* Authorization là quá trình để xác định xem người dùng được xác thực có quyền truy cập vào các tài nguyên cụ thể hay không. Nó xác minh quyền của bạn để cấp cho bạn quyền truy cập vào các tài nguyên như thông tin, cơ sở dữ liệu, file, v.v. Authorization thường được đưa ra sau khi xác thực xác nhận các đặc quyền của bạn để thực hiện. Nói một cách đơn giản hơn, nó giống như cho phép ai đó chính thức làm điều gì đó hoặc bất cứ điều gì.
* Sau khi login, một mã access Token sẽ được gửi về phía client, client sẽ sử dụng mã này để gửi về kèm theo request, server sẽ check xem mã token này có phải do server gửi hay không, nếu đúng server sẽ cho phép request này thực hiện.



### Midterm project

#### Dự án task/issue checking:

Mô phỏng 1 dự án thật ở công ty, sử dụng những kiến thức đã học trong phát triển triển các api cho dự án checking issue. Trong dự án có 2 đối tượng người dùng:

* Admin:
  + Auth
    - Đăng nhập (username, password - fixed)
  + Quản lý các project trong hệ thống,
    - Danh sách dự án (tên, tổng số task, process - số task close/total)
    - Chi tiết dự án (tên, thành viên tham gia, danh sách task trong dự án, thông số liên quan đến các trạng thái task)
    - Thêm dự án mới (tên, slug, start\_date, end\_date),
    - Sửa dự án (tên, slug, start\_date, end\_date),
    - Xoá dự án
    - Thêm người vào dự án,
    - Xoá người trong dự án
  + Quản lý người dùng,
    - Tạo invite id cho người dùng mới (chọn dự án mặc định cho họ - chỉ dùng được 1 lần, )
    - List tất cả người dùng
    - Xem chi tiết người dùng (dự án và task của họ)
    - Xoá người dùng
    - Sửa người dùng (tên, ngày sinh, email, trạng thái - active/inactive)
  + Quản lý type
    - Danh sách các type (mặc định, có bug màu đỏ, feature màu xanh)
    - Tạo mới task type (tên, màu sắc)
    - Sửa task type
    - Ẩn task type
  + Quản lý các trạng thái
    - Danh sách các trạng thái (mặc định có trạng thái close đứng cuối và new đứng đầu danh sách)
    - Tạo trạng thái mới (tên, thứ tự)
    - Sửa trạng thái (tên, thứ tự)
    - Ẩn trạng thái
  + Quản lý các độ ưu tiên
    - Danh sách các độ ưu tiên
    - Tạo mới độ ưu tiên (tên, thứ tự)
    - Sửa độ ưu tiên (tên, thứ tự)
    - Ẩn trạng thái
  + Quản lý danh sách task/issue
    - Danh sách tất cả các task/issue có trong hệ thống (chia theo từng object của trạng thái và sắp xếp theo thứ tự độ ưu tiên)
    - Tạo mới task (dự án, tên, type, độ ưu tiên, trạng thái - mặc định là new, assignee - mặc định là me, start date - phải nằm trong thời gian của dự án, end date - phải nằm trong thời gian của dự án)
    - Sửa task (dự án, tên, type, độ ưu tiên, trạng thái, assignee, start date - phải nằm trong thời gian của dự án, end date - phải nằm trong thời gian của dự án)
    - Xoá task
* User:
  + Auth
    - Tạo tài khoản theo invite (username, password, inviteid, tên, ngày sinh, email)
    - Đăng nhập (username, password, tài khoản bị inactive không thể đăng nhập thành công)
  + Join project
    - Danh sách các project được join
    - Chi tiết project (tên, tổng số task, process - số task close/total, start\_date, end\_date)
      * Danh sách tất cả các task/issue có trong project (chia theo từng object của trạng thái và sắp xếp theo thứ tự độ ưu tiên)
      * Tạo mới task (dự án - mặc định dự án hiện tại, tên, type, độ ưu tiên, trạng thái - mặc định là new, assignee - mặc định là me, start date - phải nằm trong thời gian của dự án, end date - phải nằm trong thời gian của dự án)
      * Sửa task (tên, type, độ ưu tiên, trạng thái, assignee, start date - phải nằm trong thời gian của dự án, end date - phải nằm trong thời gian của dự án)
      * Xoá task
  + Quản lý task của bản thân
    - Danh sách task của bạn thân
    - Tạo mới task (dự án, tên, type, độ ưu tiên, trạng thái - mặc định là new, assignee - mặc định là me, start date - phải nằm trong thời gian của dự án, end date - phải nằm trong thời gian của dự án)
    - Sửa task (tên, type, độ ưu tiên, trạng thái, assignee, start date - phải nằm trong thời gian của dự án, end date - phải nằm trong thời gian của dự án)
    - Xoá task

## Section 3

Trước khi bắt đầu session này, hãy chọn 1 chủ đề ( project ) để áp dụng những gì đã học ở từng mục: Chủ đề ví dụ ( quản lý nhà kho, quản lý thư viện, Website sản phẩm, …. )

Các chủ đề có thể đa dạng tuỳ theo ý tưởng của mình.

Yêu cầu:

- Tối thiểu 3 modules

- Có Authen/Author

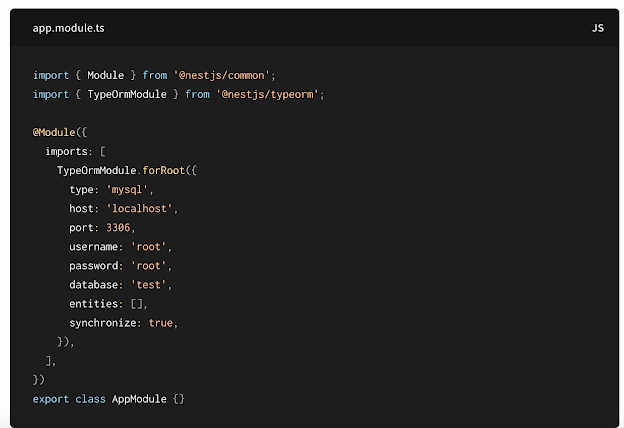
Bài tập : mỗi phần sẽ áp dụng các thành phần của nestjs vào project cụ thể

### DB connection

#### Tích hợp TypeOrm

npm install --save @nestjs/typeorm typeorm mysql2

Sau khi quá trình cài đặt hoàn tất, chúng ta có thể nhập TypeOrmModule vào AppModule gốc.



**Lưu ý**: Cài đặt synchronize: true không nên sử dụng cho production, nếu không có thể mất dữ liệu.

Phương thức forRoot hỗ trợ các thuộc tính qua hàm createConnection() từ gói TypeORM. Thêm nữa, có thể cấu hình các thuộc tính dưới:

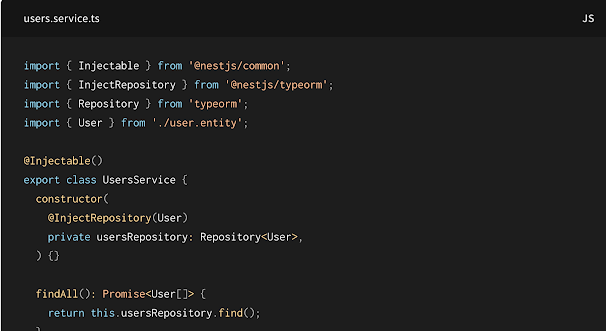
|  |  |
| --- | --- |
| retryAttempts | Số lần cố gắng kết nối với cơ sở dữ liệu (default: 10) |
| retryDelay | Độ trễ giữa các lần thử lại kết nối (mili giây) (default: 3000) |
| autoLoadEntities | Nếu đúng, các thực thể sẽ được tải tự động (default: false) |
| keepConnectionAlive | Nếu true, kết nối sẽ không bị đóng khi tắt ứng dụng (default: false) |

#### Repository pattern



Sau đó chúng ta sẽ khai báo UserEntity này vào TypeOrm Config

Sử dụng phương thức **forFeature** để xác định Entity sẽ dùng trong module. Hoặc có thể đưa **UsersRepository** vào **UsersService** bằng cách sử dụn decorator @**InjectRepository** ():



#### Relations

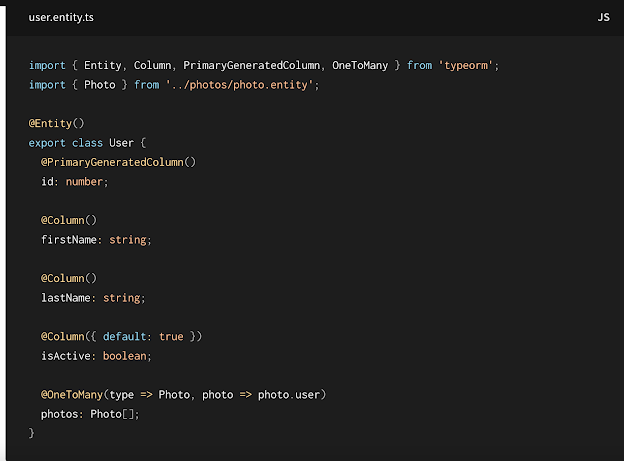
Relation là sự liên kết được thiết lập giữa hai hoặc nhiều bảng. Các mối quan hệ dựa trên các trường chung từ mỗi bảng, thường liên quan đến khóa chính và khóa ngoài.

Có ba loại quan hệ:

* One-to-One
* Many-to-One/One-to-many
* Many-to-Many

Ví dụ:

1 User có nhiều ảnh ( OneToMany ), thì ta khai báo như sau



#### Transactions

* Transaction là một tiến trình xử lý có xác định điểm đầu và điểm cuối, được chia nhỏ thành các operation (phép thực thi) , tiến trình được thực thi một cách tuần tự và độc lập các operation đó theo nguyên tắc hoặc tất cả đều thành công hoặc một operation thất bại thì toàn bộ tiến trình thất bại.
* Nếu việc thực thi một operation nào đó bị fail đồng nghĩa với việc dữ liệu phải rollback về trạng thái ban đầu.



### 

#### Migrations

* Migrations cung cấp một cách để cập nhật từng bước database schema để giữ cho nó đồng bộ với mô hình dữ liệu của ứng dụng
* Nếu bạn sữ dụng synchronize: true thì không cần phải migrations vì nó sẽ tự động đồng bộ dữ liệu cho bạn nhưng nó lại khá rủi ro

Thêm 3 script vào file package.json:

*"migrate:create": "ts-node ./node\_modules/typeorm/cli.js migration:create -n",*

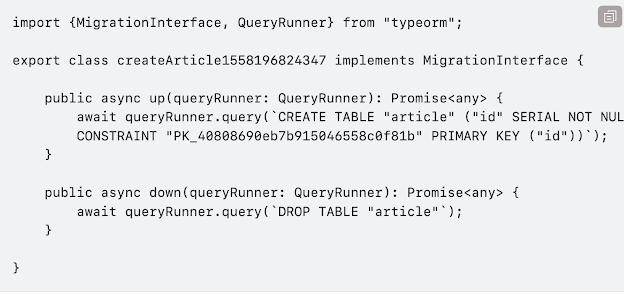
*"migrate:up": "ts-node ./node\_modules/typeorm/cli.js migration:run",*

*"migrate:down": "ts-node ./node\_modules/typeorm/cli.js migration:revert"*

Ví dụ ta có 1 Entity, sau đó chạy câu lệnh

*npm run migrate:create create*

Thu được:



Cuối cùng chạy lệnh migrate:up để sinh ra bảng trong database:

*npm run migrate:up*

#### Multiple databases

Một số dự án yêu cầu kết nối nhiều cơ sở dữ liệu,

Để làm được điều đó, ta cần đặt tên cho các kết nối ví dụ như sau:



Sau khi khai báo connection với tên của nó thì khi import feature hoặc **InjectionRepository**, **InjectConnection**, **InjectEntity** ta cần khai báo kèm name đằng sau





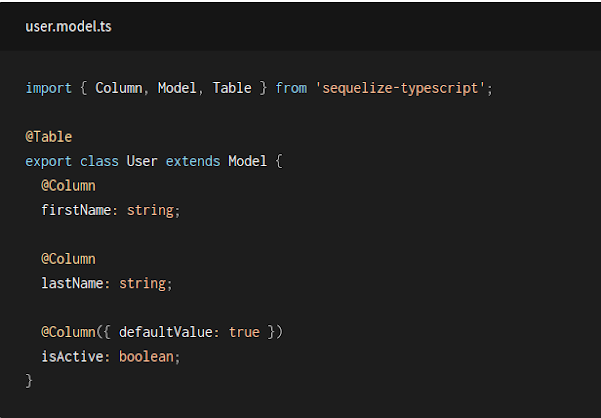
#### Tích hợp Sequelize

#### Cài đặt

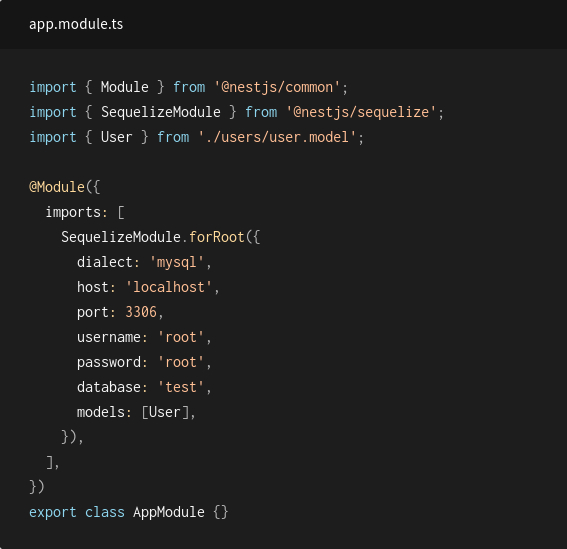
* npm install --save @nestjs/sequelize sequelize sequelize-typescript mysql2
* npm install --save-dev @types/sequelize
* Sau khi cài đặt các thư viện cần thiết, có thể import SequelizeModule vào trong module root AppModule
* 
* method forRoot hỗ trợ tất cả các thuộc tính kết nối được expose ra từ Sequelize. Có thêm một số các thuộc tính bổ sung nữa như dưới đây:
* retryAttempts : Số lần cố gắng kết nối đến csdl ( mặc định là 10 )
* retryDelay : Thời gian delay giữa các lần kết nối ( mặc định 3000ms )
* autoLoadModels : Có cho phép tự động load các model hay ko ( mặc định là false )
* synchronize : Tự động load các model để đồng bộ ( mặc định là true )
* keepConnectionAlive: nếu set = true, kết nối sẽ không bị đóng khi tắt ứng dụng

#### Xây dựng model cho các table

* Sequelize sẽ được triển khai theo mô hình Active Record. với mô hình này, sẽ sử dụng các class model để tương tác trực tiêps với db. Ví dụ sẽ tạo model User theo mô hình bên dưới



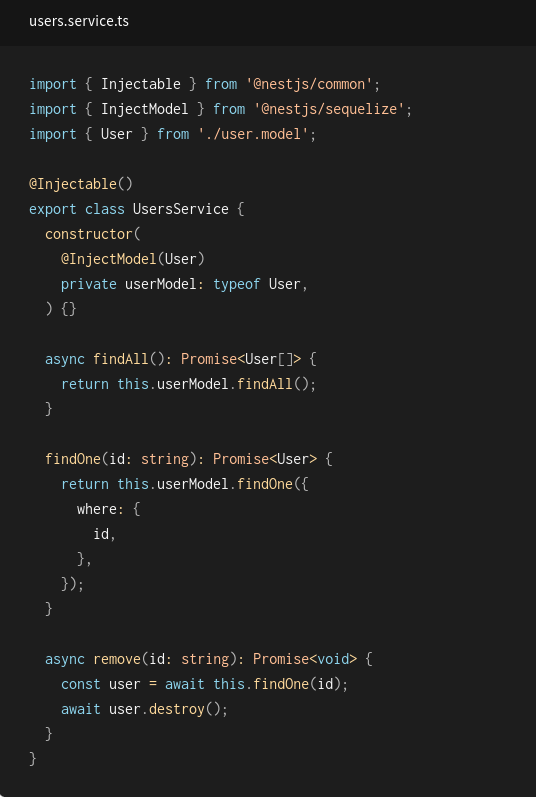
* model user này nằm trong folder user. Đường dẫn folder này chứa tất cả các file có liên quan đến UserModule. File này có thể tạo ở vị nào tùy ý nhưng được khuyến nghị được tạo ở folder module tương ứng
* Để có thể sử dụng User model, cần phải cho Sequelize biết được nó thông qua việc insert model trong thuộc tính models của module forRoot()



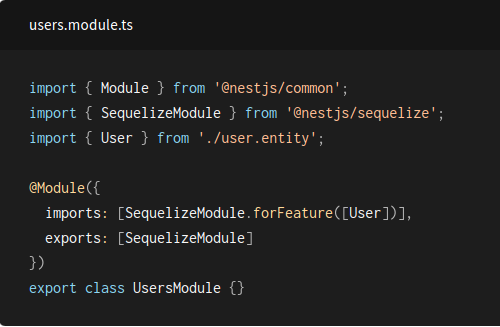
* Tiếp theo sẽ chỉnh sửa đến UsersModule



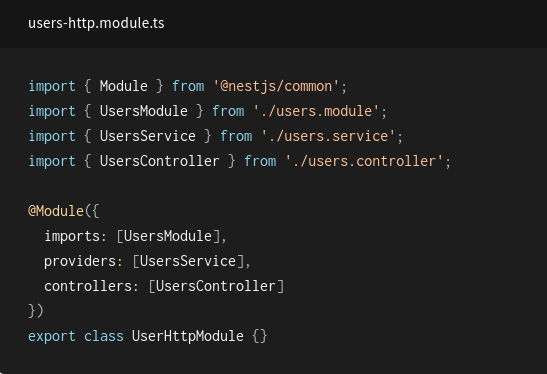
* Sử dụng module forFeature() để xác định được model sẽ được đăng ký trong phạm vi hiện tại. Khi đó, có thể inject UserModel vào trong UserService thông qua việc sử dụng decorator @Injectmodel()



* Nếu muốn sử dụng repository bên ngoài module thì có thể export nó ra bên ngoài, có thể làm như sau



* Bây giờ có thể import UserModule trong UserHttpModule, có thể sử dụng @InjectModel(User) trong provider của các module sau đó



#### 

#### Xây dựng quan hệ các model

* Có thể tạo quan hệ giữa các model từ các mối quan hê 1:1, 1:n, n:n. Để xác định được mối quan hệ giữa các model, sử dụng các decorator tương ứng. ví dụ để xác định mỗi user sẽ có nhiều photo, sử dụng decorator @HasMany() như dưới đây



#### Tự động load các model

* Có thể thêm thủ công bằng việc add các model vào trong thuộc tính models của method forRoot(). Để có thể giải quyết việc thêm thủ công như vậy có thể dùng thuộc tính autoLoadModels và synchronize để làm việc đó. ví dụ :



* Để sử dung được cách trên thì mọi model cần phải được đăng ký thông qua method forFeature(), khi đó nó sẽ tự động được add vào mảng models

#### 

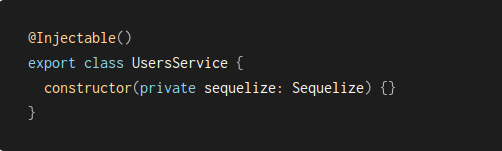
#### 

#### 

#### 

#### Transaction

* Đầu tiên cần phải inject Sequelize vào bên trong class như bình thường



* bây giờ có thể tạo transaction như bên dưới

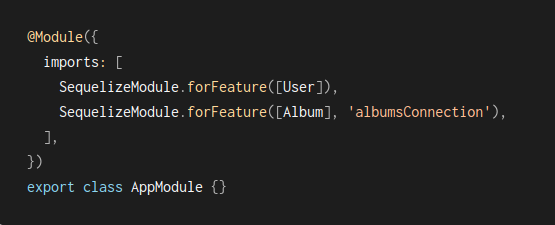


Kết nối nhiều database

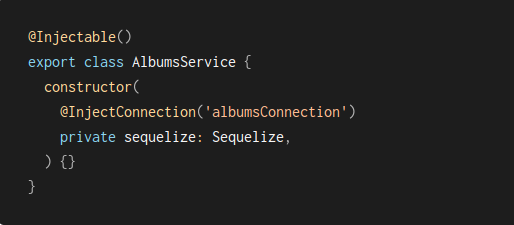
* Trong một vài dự án yêu cầu có thể cần kết nối đến nhiều db, để tạo được kết nối đến nhiều db, tên của các connection cần phải được đặt và bắt buộc. Nếu như không đặt tên cho connection thì tên mặc định sẽ là default. ví dụ như bên dưới :



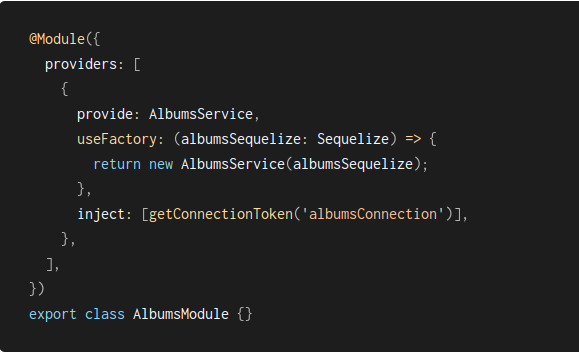
* ở thời điểm này, model User và Album đã được đăng ký. với setup này thì có thể gọi SequelizeModule.forFeature() và decorator JnjectModel() để kết nối đến chúng và sử dụng. Nếu như k có tên kết nối thì tên default sẽ được sử dụng



* Cũng có thể inject đối tượng Sequelize để kết nối



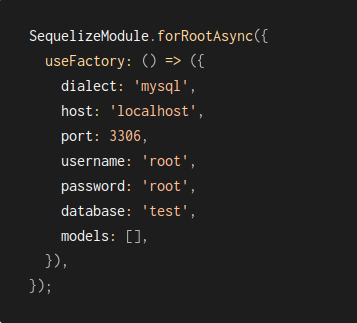
* Nó cũng có thể inject bất cứ đối tượng Sequelize nào trong providers



#### 

#### Kết nối bất đồng bộ

* Có thể lựa chọn tùy chọn kết nối không đồng bộ thay vì đồng bộ như mặc định. Trong trường hợp này sử dụng mothod forRootAsync(), ví dụ như bên dưới:



### Nestjs

#### Định nghĩa

Nest tận dụng các tính năng ngôn ngữ mới nhất, vì vậy, để sử dụng nó với JavaScript thông thường, chúng ta cần có trình biên dịch Babel.

Chúng ta chủ yếu sẽ sử dụng TypeScript trong các ví dụ mà Chúng ta cung cấp, nhưng chúng ta luôn có thể chuyển các đoạn mã sang cú pháp JavaScript thuần (chỉ cần nhấp để chuyển nút ngôn ngữ ở góc trên bên phải của mỗi đoạn mã).

Hãy đảm bảo rằng Node.js (> = 10.13.0, ngoại trừ v13) được cài đặt

#### Setup

**$ npm i -g @nestjs/cli**

**$ nest new project-name**

Có hai nền tảng HTTP được hỗ trợ ngay lập tức: express và fastify.

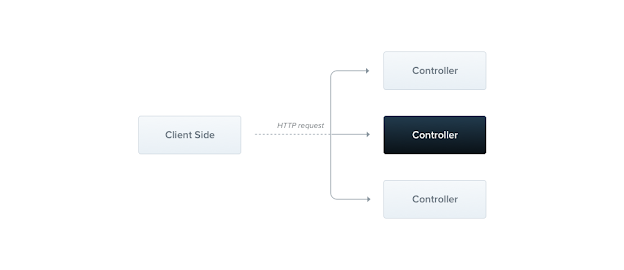
|  |  |
| --- | --- |
| platform-express | [Express](https://expressjs.com/) là một framework web nổi tiếng cho node. Đó là một thư viện đã được kiểm chứng, sẵn sàng cho việc sản xuất với rất nhiều tài nguyên do cộng đồng thực hiện. @nestjs/platform-express là gói được sử dụng mặc định. Rất nhiều người được phục vụ với Express, và không cần thêm thao tác nào để kích hoạt nó. |
| platform-fastify | [Fastify](https://www.fastify.io/) là framework có hiệu suất cao và chi phí thấp, tập trung vào cung cấp hiệu quả và tốc độ tối đa. |

Sau khi tạo project thành công, chạy lệnh để bắt đầu phát triển dự án

**$ npm run start**

Câu lệnh bắt đầu ứng dụng với lắng nghe cổng HTTP được định nghĩa trong file src/main.ts. Một khi ứng dụng chạy, mở trình duyệt của chúng ta, nó sẽ chuyển đến http://localhost:3000/. Chúng ta sẽ thấy thông điệp Hello world! .

#### Controller



Mục đích chính của controllers là tiếp nhận các request cụ thể từ ứng dụng. Cơ chế routing sẽ chỉ định các controllers nào cần tiếp nhận và xử lý requests. Thông thường, mỗi controller sẽ có nhiều routes, mỗi routes chịu trách nhiệm xử lý các actions cụ thể.

Để khởi tạo nhanh các **CRUD controllers** đi kèm với bộ validation có sẵn, chúng ta có thể sử dụng đoạn CLI sau:

$ nest g resource your-resource-name

##### Routing

Chúng ta có thể khai báo một Controller chúng tag cách tạo một class kèm theo decorator @Controller(). Chúng ta cũng có thể group các controllers lại với nhau thông qua việc chỉ định optional path cho decorator @Controller. Điều này cũng sẽ giúp chúng ta giảm thiểu code lặp không cần thiết.

// cats.controller.ts

import { Controller, Get } from '@nestjs/common';

@Controller('cats')

export *class* CatsController {

@Get()

findAll(): *string* {

return 'This action returns all cats'

}

}

Chúng ta có thể tạo một controller chúng tag CLI như sau

$ nest g controller your-controller-name

Decorator @Get() được khai báo phía trên method findAll() giúp Nest xác định và tiến hành điều hướng xử lý requests.

Bởi vì chúng ta đã khai báo prefix path cho controller là cats, và chúng ta cũng để trống phần path trong decorator @Get(), nên khi chúng ta truy cập link http://localhost:3000/cats qua phương thức GET, Nest sẽ điều hướng xử lý tới method findAll() mà chúng ta đã khai báo trong controller.

Theo mặc định thì method này sẽ trả về dữ liệu kèm theo Http Status Code **200**. Chúng ta cũng có thể thay đổi chúng nếu muốn.

##### Thông tin Request

Trong vài trường hợp, chúng ta cần lấy một số thông tin chi tiết từ request phía client. Nest cung cấp cho chúng ta decorator @Req() để lấy thông tin từ client request. Biến này sẽ là Request (nếu chúng ta sử dụng platform Express), hoặc FastifyRequest (nếu chúng ta sử dụng platform Fastify),…

import { Controller, Get, Req } from '@nestjs/common';

import { Request } from 'express';

@Controller('cats')

export *class* CatsController {

@Get()

findAll(@Req() request: Request): *string* {

return 'This action returns all cats'

}

}

Request Object của Nest chứa rất nhiều thông tin, và trong hầu hết các trường hợp chúng ta cũng không cần phải lấy các thông tin này một cách thủ công. Do đó, Nest cung cấp cho chúng ta một số request decorators khác giúp chúng ta dễ dàng thao tác và xử lý hơn.

* @Request(), @Req() ==> req
* @Response(), @Res() ==> res

Lưu ý: khi sử dụng @Response() hoặc @Res(), chúng ta cần tự quản lý việc phản hồi, sẽ không làm điều đó cho chúng ta. Ví dụ, chúng ta cần gọi res.json()... hoặc res.send()..., nếu không hệ thống sẽ bị treo vĩnh viễn.

Hoặc chúng ta cũng có thể sử dụng thiết lập @Res({ passthrough: true }). Lúc này, chúng ta không cần phải tự quản lý việc phản hồi, mà chỉ tập trung vào việc chỉnh sửa một số thông tin khác trên response. Phần còn lại, Nest sẽ làm giúp chúng ta.

* @Next() ==> next
* @Session() ==> req.session
* @Param(key?: string) ==> req.params / req.params[key]
* @Body(key?: string) ==> req.body / req.body[key]
* @Query(key?: string) ==> req.query / req.query[key]
* @Headers(name?: string) ==> req.headers / req.headers[name]
* @Ip() ==> req.ip
* @HostParam() ==> req.hosts

##### HTTP methods

Nest cung cấp các một số decorators giúp thiết lập routing qua HTTP methods.

* @Get() dùng để lấy dữ liệu.
* @Post() cơ chúng ta là tạo mới thông tin.
* @Put() thường được dùng để cập nhật thông tin một chúng ta ghi. Nên nhớ là nó sẽ thay thế chúng ta ghi chúng tag nguyên cục data chúng ta truyền vào.
* @Patch() cũng được dùng để cập nhật thông tin. Nhưng khác ở chỗ, nó chỉ cập nhật một vài fields được yêu cầu thay vì toàn bộ.
* @Delete() xóa dữ liệu.
* @All() phương thức này chấp nhận mọi HTTP methods.

Ngoài ra còn có @Options(), @Heap()…

##### HTTP Status Code

Theo mặc định, repsonse code luôn luôn là **200**, đối với các request **POST** là **201**. Chúng ta có thể thay đổi nó một cách dễ dàng thông qua decorator @HttpCode(...).

@Post()

@HttpCode(204)

create() {

return 'This action adds a new cat'

}

Đôi lúc, response code của chúng ta không cố định, mà phụ thuộc vào các yếu tố khác nhau. Trong trường hợp đó, chúng ta có thể chỉ định chúng một cách dynamic thông qua @Res().

Chúng ta cần import decorator HttpCode từ package @nestjs/common.

##### Headers

Để tùy chỉnh một Header, chúng ta có thể dùng decorator @Header()

@Post()

@Header('Cache-Control', 'none')

create() {

return 'This action adds a new cat'

}

Hoặc thông qua @Res()

@Post()

create(@Res() res) {

res

.header('Cache-Control', 'none')

.send('This action adds a new cat')

}

Chúng ta có thể import decorator Header từ package @nestjs/common.

##### Redirect

Để redirect tới một URL cụ thể, chúng ta có thể sử dụng decorator @Redirect() hoặc thông qua @Res(), sau đó call trực tiếp res.redirect().

Decorator @Redirect() nhận 2 optional params là url: string và statusCode: number. Giá trị mặc định của statusCode là **302**.

@Get()

@Redirect('https://duypt.dev', 301)

Để thực hiện redirect một cách dynamic, chúng ta có thể sử dụng cách sau

@Get('docs')

@Redirect('https://docs.nestjs.com', 301)

getDocs(@Query('version') version) {

if (version && version === '5') {

return { url: 'https://docs.nestjs.com/v5/', statusCode: 301 }

}

}

Giá trị trả về sẽ **override** giá trị chúng ta truyền vào decorator @Redirect().

##### Route parameters

Khai báo routes với path tĩnh sẽ không hoạt động khi chúng ta cần truyền **dynamic data** trực tiếp vào URL.

Để khai báo một route chấp nhận dynamic params, chúng ta có thể làm như sau

@Get(':id')

findOne(@Param('id') id): string {

return `This action returns a #${id} cat`

}

Nếu chúng ta không khai báo name cho decorator @Param(), nó sẽ trả về toàn bộ route params nếu tồn tại.

Chúng ta cần import decorator Param từ package @nestjs/common.

###### 

##### Request scopes

Đối với một số chúng ta có nền tảng kiến thức từ một số ngôn ngữ lập trình khác (như PHP…), có thể họ sẽ khá bất ngờ khi mà hầu hết mọi thứ đều được “**xài chung**” trong các requests. Chúng ta có một cái connection tới database, một số singleton services được dùng toàn cục…

Hãy nhớ rằng, NodeJS không tuân theo mô hình **Multi-Threaded Stateless Model** (nơi mà mỗi yêu cầu được xử lý trên một thread riêng biệt). Do đó, việc sử dụng singleton hoàn toàn an toàn với Nest.

Tuy nhiên, đôi lúc chúng ta mong muốn một số service chỉ hoạt động trong phạm vi một request cụ thể nào đó để tránh việc chồng chéo lên nhau. Đó là lúc chúng ta cần quan tâm tới r**equest scopes**.

Chúng ta sẽ đi sâu hơn về phần này trong các phần tiếp theo.

##### Bất đồng bộ (Asynchronicity)

Hầu hết các công việc thao tác với dữ liệu hoặc hệ thống khác đều là bất đồng bộ. NestJS cũng hỗ trợ và hoạt động rất tốt với async.

Tất cả các async functions đều trả về một Promise. Nest sẽ tự giải quyết phần xử lý cho chúng ta.

@Get()

async findAll(): *Promise*<*any*[]> {

return []

}

Ngoài ra, Nest còn hỗ trợ một framework xử lý bất đồng bộ cực kỳ mạnh mẽ – **RxJS** theo cơ chế **Observable streams** hiện đại. Nest sẽ **tự động subscribe** để lấy về giá trị cuối cùng khi **stream completed**.

@Get()

findAll(): Observable<*any*[]> {

return of([])

}

##### Truyền nhận dữ liệu qua DTO

Nest hỗ trợ chúng ta đọc dữ liệu truyền lên từ client request thông qua decorator @Body().

Chúng ta cần lưu ý là theo chuẩn HTTP, các requests sử dụng method **POST**, **PUT**, **PATCH** mới có thể đính kèm payload, do đó decorator @Body() chỉ hoạt động với các HTTP methods này.

**DTO** (**Data Transfer Object**) – là cách triển khai một design pattern rất phổ biến – **Transfer Object Pattern**. Chúng đơn giản là các object xác định kiểu dữ liệu sẽ được gửi đi hoặc nhận về, và có thể được serialize khi truyền qua mạng. Chúng không – và cũng không nên có các logic xử lý bên dưới.

Nest hỗ trợ chúng ta khai báo DTO qua interface hoặc các class.

* Class là một phần trong chuẩn Javascript ES6. Do đó, chúng được giữ lại sau khi compiled.
* Trong khi đó interface chỉ là cú pháp của Typescript, chúng không tồn tại, và cũng sẽ bị xóa khỏi code Javascript sau khi compiled. Do đó, Nest không thể tham chiếu đến chúng trong lúc chạy.
* Thêm một vài điểm nhấn quan trọng khi chúng ta sử dụng class:
  + Rất dễ dàng thực hiện validate dữ liệu đầu cuối thông qua class-validator.
  + Có thể transform dữ liệu thông qua các Pipes. Bởi class là tồn tại, do đó chúng ta có thể can thiệp vào các metadata của chúng trong thời gian chạy.

//create-cat.dto.ts

export *class* CreateCatDto {

name: *string*

age: *number*

breed: *string*

}

//cats.controller.ts

@Post()

async create(@Body() createCatDto: CreateCatDto) {

return 'This action adds a new cat'

}

Chúng ta cũng có thể thêm một vài validation như thế này

//create-cat.dto.ts

export *class* CreateCatDto {

@IsNotEmpty()

name: *string*

@IsInt()

@Optional()

age: *number*

@MinLength(5)

breed: *string*}

#### Provider

Providers đại diện cho các class Service. Điều đó có nghĩa là các providers sẽ được **inject** vào những nơi cần sử dụng thông qua DI. Một số class cơ chúng ta trong Nest được xem là provider như

* Service
* Factory
* Repository
* Helper
* …

##### Dependency Injection

Trong AppService, chúng ta định nghĩa nó là một dependency chúng tag cách sử dụng decorator @Injectable().

Ngoài ra, AppService được inject vào AppController thông qua constructor.

constructor(private readonly appService: AppService) {}

Việc sử dụng **Access Modifier** (public – protected – private) trong constructor là một cách viết tắt giúp chúng ta có thể khai báo và khởi tạo property trong class cùng một lúc.

Đoạn code trên tương đương với cách viết như sau:

@Controller()

export *class* AppController {

private readonly appService: AppService

constructor(appService: AppService) { this.appService = appService }

}

Một điều quan trọng nữa là các chúng ta cần **register** AppService trong Nest module để nó **resolve** và thực hiện **inject** AppService đúng cách.

Quá trình cũng rất đơn giản. Chúng ta chỉ cần chỉnh sửa file Nest module, sau đó thêm AppService vào mảng providers của decorator @Module().

import { Module } from '@nestjs/common';

import { AppController } from './app.controller';

import { AppService } from './app.service';

@Module({ imports: [], controllers: [AppController], providers: [AppService], })

export *class* AppModule {} ﻿

Nest cung cấp sẵn một hệ thống IoC Container riêng giúp thực hiện DI một cách dễ dàng. Chúng ta sẽ tóm tắt cơ chế hoạt động của nó như sau

* Trong app.service.ts, decorator @Injectable() khai báo cho IoC Container biết AppService là một dependency được quản lý bởi IoC Container.
* Trong app.controller.ts, controller AppController khai báo nó cần dependency AppService thông qua constructor.
* Trong app.module.ts, Nest tạo ra một **token** liên kết với AppService từ tập tin app.service.ts. Việc đăng ký được thực hiện thông qua mảng providers của decorator @Module().
* Khi Nest khởi tạo AppController, nó sẽ tìm kiếm tất cả các dependencies mà AppController yêu cầu – ở đây là AppService.
* IoC Container lúc này sẽ kiểm tra tất cả providers đã được **register** trong Nest module và tìm AppService thông qua AppService **token** đã được đăng ký trước đó.
* Sau khi tìm thấy, nó sẽ khởi tạo (hoặc lấy về nếu đã tồn tại) instance của AppService, sau đó khởi tạo class AppController, đồng thời inject AppService vào AppController thông qua constructor.
* Nếu không tìm thấy, Nest sẽ báo lỗi cho chúng ta biết.

Ngoài ra, IoC Container còn thực hiện analysis các dependencies (Nest gọi nó là tạo **Dependency Graph**). **Dependency Graph** bảo đảm các dependencies được **resolve theo thứ tự** – về cơ chúng ta là “**từ dưới lên**“.

##### Providers customize

Đây là cách khai báo đơn giản nhất, có thể sử dụng trong hầu hết các trường hợp.

@Module({ imports: [], controllers: [AppController], providers: [AppService], })

Chúng ta có thể truyền trực tiếp các class name vào trong mảng providers. Cú pháp này là cách viết tắt của cách viết đầy đủ như sau:

providers: [ { provide: AppService, useClass: AppService, } ]

Nest hỗ trợ một số phương thức để thực hiện resolve dependencies:

* useClass
* useValue
* useFactory
* useExisting

useClass

Chúng ta chỉ cần khai báo tên **class** chúng ta muốn sử dụng làm instance. Hãy nhớ, chỉ có các class đăng ký @Injectable() mới có thể sử dụng làm instance theo cách này.

useValue

Nest sẽ resolve dependency chúng tag chính **instance** hoặc **value** mà chúng ta khai báo. Cách này thường được dùng để viết mock test.

import { CatsService } from './cats.service';

const mockCatsService = { /\* mock implementation ... \*/ };

@Module({ imports: [CatsModule], providers: [ { provide: CatsService, useValue: mockCatsService, }, ], })

export *class* AppModule {}

useFactory

Cú pháp này giúp chúng ta các provider một cách dynamic. Dependency sẽ được resolve chúng tag giá trị của hàm useFactory(). Chúng ta cũng có thể inject dependencies vào các factory phức tạp.

* Chúng ta có thể truyền các optional params vào factory function
* Nếu truyền params vào factory function, chúng ta cần khai báo thêm các dependencies cho factory thông qua property inject.

const connectionFactory = {

provide: 'CONNECTION',

useFactory: (optionsProvider: OptionsProvider) => {

const options = optionsProvider.get();

return new DatabaseConnection(options); },

inject: [OptionsProvider], };

@Module({ providers: [connectionFactory], }) e

xport *class* AppModule {}

useExisting

Phương thức này giúp chúng ta alias các providers đã tồn tại.

@Injectable()

*class* LoggerService { /\* implementation details \*/ }

const loggerAliasProvider = { provide: 'AliasedLoggerService', useExisting: LoggerService, };

@Module({ providers: [LoggerService, loggerAliasProvider], })

export *class* AppModule {}

Ở ví dụ trên, string token `AliasedLoggerService` là một alias của LoggerService. Nếu scope của cả 2 provider này đều là SINGLETON, thì chúng sẽ trả về cùng một instance.

##### Non-class provider tokens

Như đã đề cập ở ví dụ trước đó, Nest cũng cho phép chúng ta register **DI** tokens chúng tag **string** hoặc Symbol.

@Module({ providers: [

{

provide: 'CAT\_REPOSITORY',

useValue: new CatRepository(),

},

],

})

export *class* AppModule {}

Sau đó sử dụng decorator Inject() ở nơi cần sử dụng.

@Controller()

export *class* AppController {

constructor(@Inject('MY\_REPOSITORY') private readonly repository: CatRepository {} ... }

##### Export Provider

Giả sử chúng ta có 2 module A và B. Nếu chúng ta muốn một provider khai báo trong module A có thể được dùng bởi module B, đây là lúc chúng ta cần export provider đó từ module A, sau đó import module A vào trong module B để sử dụng.

const connectionFactory = { provide: 'CONNECTION',

useFactory: (optionsProvider: OptionsProvider) => {

const options = optionsProvider.get();

return new DatabaseConnection(options); },

inject: [OptionsProvider], };

@Module({ providers: [connectionFactory],

exports: ['CONNECTION'], })

export *class* AppModule {}

Thực ra thì export luôn cái provider object cũng được, chả sao cả.

@Module({ providers: [connectionFactory], exports: [connectionFactory], })

export *class* AppModule {}

#### Module

@Module() decorator nhận một đối tượng duy nhất có thuộc tính mô tả module:

|  |  |
| --- | --- |
| providers | Các Provider sẽ được khởi tạo bởi injector Nest và ít nhất có thể được chia sẻ trên module này |
| controllers | Tập hợp các controller được xác định trong module này phải được khởi tạo |
| imports | Danh sách các module được imported để xuất các providers được yêu cầu trong module này. |
| exports | Tập hợp con các providers được cung cấp bởi module này và sẽ có sẵn trong các module khác mà import module này. |

Module đóng gói các providers theo mặc định. Điều này có nghĩa là không thể inject providers mà ko trực tiếp là 1 phần của module hiện tại cũng như đã exported từ các module đã imported. Do đó, bạn có thể coi các providers đã exported từ một module là giao diện công khai của module hay API.

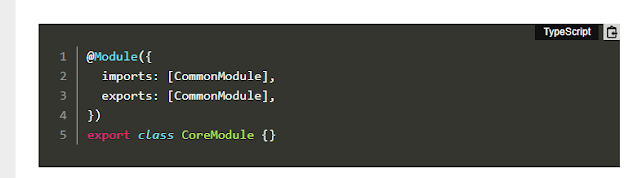
##### Shared modules

Trong Nest, các module là singleton theo mặc định, và do đó bạn có thể chia sẻ cùng một thực thể của bất kỳ provider nào giữa nhiều module một cách dễ dàng.



##### Module re-exporting

Module có thể export các providers nội bộ của chúng. Ngoài ra, họ có thể re-import các module mà họ đã import. Trong ví dụ bên dưới, CommonModule được imported và export từ CoreModule, làm cho nó có sẵn cho các module khác nhập module này.



##### Dependency injection

Một class module cũng có thể inject các providers (ví dụ: cho mục đích cấu hình):



##### Global modules

Khi bạn muốn cung cấp một tập hợp các providers nên có sẵn ở mọi nơi ngay lập tức (ví dụ: helpers, database connections, v.v.), hãy đặt module toàn cục với @Global() decorator.



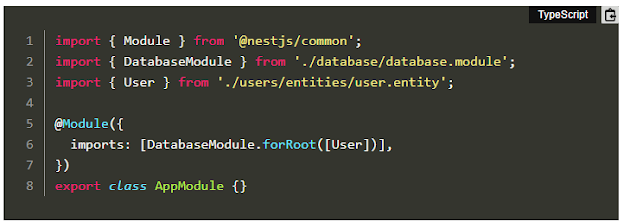
##### Dynamic modules

Hệ thống module Nest bao gồm một tính năng mạnh mẽ được gọi là module động. Tính năng này cho phép bạn dễ dàng tạo các module có thể tùy chỉnh có thể đăng ký và cấu hình động các providers. Các mô-đun động được đề cập rộng rãi [ở đây](https://docs.nestjs.com/fundamentals/dynamic-modules). Trong chương này, chúng tôi sẽ trình bày tổng quan ngắn gọn để hoàn thành phần giới thiệu về các module.

Sau đây là một ví dụ về định nghĩa mô-đun động cho DatabaseModule:



DatabaseModule có thể được nhập và cấu hình theo cách sau:



Nếu bạn muốn lần lượt xuất lại một mô-đun động, bạn có thể bỏ qua lệnh gọi phương thức forRoot() trong mảng exports:



#### Middleware

Middleware là một hàm được gọi trước khi tới handler route. Các hàm middleware có quyền truy cập vào các object request và response cũng như hàm middleware next() trong chu trình request-response của ứng dụng. Hàm middlware next thường được ký hiệu chúng tag một biến có tên là next.

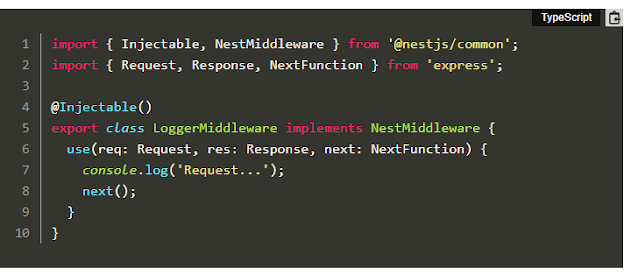


Nest middleware, theo mặc định, tương đương với express middleware. Mô tả sau đây từ tài liệu express chính thức mô tả các khả năng của middleware:

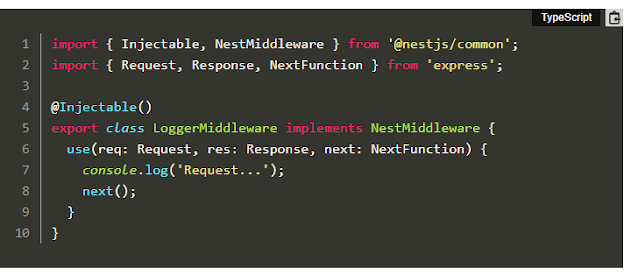
Các hàm Middleware có thể thực hiện các nhiệm vụ sau:

* Thực thi bất kỳ mã nào.
* Thực hiện các thay đổi đối với request và response object.
* Kết thúc chu kỳ request-response.
* Gọi hàm middleware tiếp theo trong ngăn xếp.
* Nếu hàm middleware hiện tại không kết thúc chu kỳ request-response, nó phải gọi next() để chuyển quyền điều khiển cho hàm middleware tiếp theo. Nếu không, request sẽ bị treo.

Chúng ta triển khai middleware Nest tùy chỉnh trong một hàm hoặc trong một class với @Injectable() decorator. Lớp nên implement interface NestMiddleware, trong khi hàm này không có bất kỳ yêu cầu đặc biệt nào. Hãy bắt đầu chúng tag cách triển khai một tính năng middleware đơn giản chúng tag cách sử dụng phương thức lớp.



Không có chỗ cho middleware trong @Module() decorator. Thay vào đó, Chúng ta thiết lập chúng chúng tag phương thức configure() của lớp mô-đun. Các mô-đun bao gồm middleware phải triển khai interface NestModule. Hãy thiết lập LoggerMiddleware ở cấp AppModule.



##### 

##### 

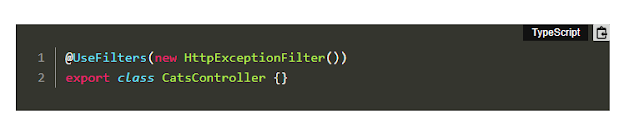
##### 

#### Filter

Để làm điều này, Chúng ta sẽ cần truy cập vào các đối tượng Request và Respose của nền tảng cơ chúng ta. Chúng ta sẽ truy cập đối tượng Request để có thể lấy ra url chúng ta đầu và đưa url đó vào thông tin ghi nhật ký. Chúng ta sẽ sử dụng đối tượng Response để kiểm soát trực tiếp response được gửi chúng tag cách sử dụng phương thức response.json().



@Catch(HttpException) decorator liên kết metadata bắt buộc với bộ lọc exception, cho Nest biết rằng bộ lọc cụ thể này đang tìm kiếm các exception của loại HttpException và không có gì khác. @Catch() decorator có thể nhận một tham số duy nhất hoặc một danh sách được phân tách chúng tag dấu phẩy. Điều này cho phép chúng ta thiết lập bộ lọc cho một số loại exception cùng một lúc.



Cấu trúc này thiết lập HttpExceptionFilter cho mọi route handler được xác định bên trong CatsController.

Để tạo bộ lọc phạm vi toàn cục, chúng ta sẽ thực hiện như sau:



#### Pipe : [link](https://docs.google.com/document/d/1zHX3WtpDs78Lj6ajjFXhKWcfOMNz0oWzY1DQsNPuQZI/edit?usp=sharing)

#### Guard: [link](https://docs.google.com/document/d/1S0zlJ9Kg_QW9YADBGBuESnU8P15qANZxE5JVBjVMEpQ/edit?usp=sharing)

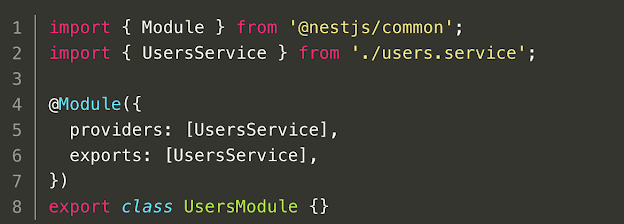
#### Interceptor: [link](https://docs.google.com/document/d/1mSuw-789CdzetKSUqv-JkG2FO8Dk3mxLKFOm1zriLc4/edit?usp=sharing)

#### Custom decorator: [link](https://docs.google.com/document/d/1Jpxt6xgUAdWviwwdAKuAN371EePsHCOSmj55MsdxPgI/edit?usp=sharing)

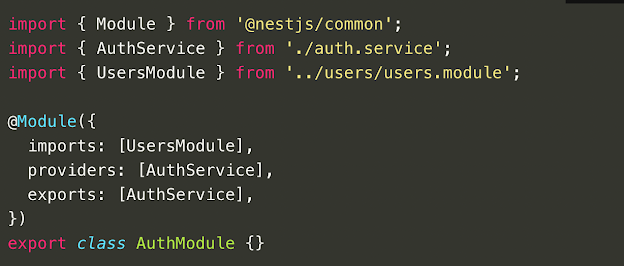
#### Dynamic module

Hầu hết các ví dụ về mã ứng dụng trong phần Overview của tài liệu đều sử dụng các mô-đun thông thường hoặc tĩnh. Mô-đun xác định các nhóm thành phần như [providers](https://docs.nestjs.com/providers) and [controllers](https://docs.nestjs.com/controllers) phù hợp với nhau như một phần mô-đun của một ứng dụng tổng thể. Chúng cung cấp bối cảnh thực thi hoặc phạm vi cho các thành phần này. Ví dụ: các providers được xác định trong một mô-đun sẽ hiển thị với các thành viên khác của mô-đun mà không cần xuất chúng. Khi một provider cần hiển thị bên ngoài một mô-đun, trước tiên nó sẽ được xuất từ mô-đun chủ của nó, sau đó được nhập vào mô-đun tiêu thụ của nó.

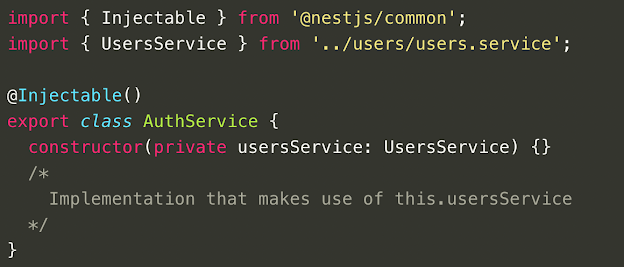
Đầu tiên, chúng tôi sẽ định nghĩa một UsersModule để provide và export một UsersService. UsersModule là mô-đun chủ cho UsersService.



Tiếp theo, chúng ta sẽ định nghĩa AuthModule, import UsersModule, làm cho các providers đã export của UsersModule có sẵn bên trong AuthModule:



Các cấu trúc này cho phép chúng tôi inject UsersService vào, chẳng hạn như AuthService được lưu trữ trong AuthModule:



Chúng tôi sẽ gọi đây là ràng buộc mô-đun tĩnh. Tất cả thông tin Nest cần để kết nối các mô-đun với nhau đã được khai báo trong mô-đun chủ và mô-đun sử dụng. Hãy giải nén những gì đang xảy ra trong quá trình này. Nest cung cấp UsersService bên trong AuthModule bằng cách:

1. Khởi tạo UsersModule, bao gồm nhập tạm thời các mô-đun khác mà chính UsersModule sử dụng và giải quyết tạm thời bất kỳ dependencies nào
2. Khởi tạo AuthModule và làm cho providers đã exported của UsersModule có sẵn cho các thành phần trong AuthModule (giống như thể chúng đã được khai báo trong AuthModule).
3. Inject một instance của UsersService vào trong AuthService.

##### Dynamic module use case

Với ràng buộc mô-đun tĩnh, không có cơ hội để mô-đun sử dụng ảnh hưởng đến cách các providers từ mô-đun chủ được định cấu hình. Vì sao vấn đề này? Hãy xem xét trường hợp chúng ta có một mô-đun mục đích chung cần hoạt động khác nhau trong các trường hợp sử dụng khác nhau. Điều này tương tự với khái niệm “plugin” trong nhiều hệ thống, trong đó một cơ sở chung yêu cầu một số cấu hình trước khi nó có thể được người tiêu dùng sử dụng.

Một ví dụ điển hình với Nest là mô-đun cấu hình. Nhiều ứng dụng thấy hữu ích khi ngoại hóa chi tiết cấu hình bằng cách sử dụng mô-đun cấu hình. Điều này giúp bạn dễ dàng thay đổi động cài đặt ứng dụng trong các lần triển khai khác nhau: ví dụ: cơ sở dữ liệu phát triển cho nhà phát triển, cơ sở dữ liệu dàn cho môi trường dàn / thử nghiệm, v.v. Bằng cách ủy quyền quản lý các tham số cấu hình cho mô-đun cấu hình, mã nguồn ứng dụng vẫn độc lập với các thông số cấu hình.

Thách thức là bản thân mô-đun cấu hình, vì nó chung chung (tương tự như “plugin”), cần được tùy chỉnh bởi mô-đun tiêu thụ của nó. Đây là lúc các mô-đun động phát huy tác dụng. Bằng cách sử dụng các tính năng của mô-đun động, chúng tôi có thể làm cho mô-đun cấu hình động để mô-đun tiêu thụ có thể sử dụng API để kiểm soát cách mô-đun cấu hình được tùy chỉnh tại thời điểm nó được nhập.

Nói cách khác, các mô-đun động cung cấp một API để nhập một mô-đun này vào một mô-đun khác và tùy chỉnh các thuộc tính và hành vi của mô-đun đó khi nó được nhập, trái ngược với việc sử dụng các ràng buộc tĩnh mà chúng ta đã thấy cho đến nay.

#### Circular dependency

Sự phụ thuộc vòng tròn xảy ra khi hai lớp phụ thuộc vào nhau. Ví dụ: lớp A cần lớp B và lớp B cũng cần lớp A. Các phụ thuộc vòng tròn có thể phát sinh trong Nest giữa các mô-đun và giữa các providers.

Mặc dù nên tránh các phụ thuộc vòng tròn nếu có thể, nhưng không phải lúc nào bạn cũng có thể làm như vậy. Trong những trường hợp như vậy, Nest cho phép giải quyết sự phụ thuộc vòng tròn giữa các providers theo hai cách. Trong chương này, chúng tôi mô tả việc sử dụng tham chiếu chuyển tiếp như một kỹ thuật và sử dụng lớp ModuleRef để truy xuất một instance provider từ vùng chứa DI như một kỹ thuật khác.

Chúng tôi cũng mô tả việc giải quyết circular dependencies giữa các mô-đun.

#### Module reference : [link](https://docs.google.com/document/d/1aDmI88oecsvSYCC9P-LwBFIpkLJXvdXJh6ax5rFZLX0/edit?usp=sharing)

#### Lifecircle event

* Ứng dụng Nest, cũng như các phần tử ứng dụng, có vòng đời do Nest quản lý. Nest cung cấp các lifecircle hook giúp thực hiện các hành động khi chạy một ứng dụng nest
* Một lifecircle event xảy ra khi một ứng dụng bắt đầu khởi chạy cho đến khi ứng dụng được kết thúc. Nest gọi các hàm đăng ký lifecircle event trong các mothod module, injectable và controller. Một số lifecircle event trong nestjs

|  |  |
| --- | --- |
| **Tên Lifecycle hook** | **Sự kiện được trigger khi gọi** |
| onModuleInit() | Được gọi khi các module trong ứng dụng nestjs đều được khởi tạo và biên dịch không gặp lỗi |
| onApplicationBootstrap() | Được gọi khi tất cả các module đều đã được khởi tạo, nhưng trước khi lắng nghe kết nối (database, … ) |
| onModuleDestroy()\* | Được gọi sau khi nhận tín hiệu kết thúc ứng dụng |
| beforeApplicationShutdown()\* | Được gọi khi tất cả các tiến trình trong mothod monMuduleDestroy() được hoàn thành, tất cả các kết nối đến các dịch vụ database, cache, queue, … được đóng |
| onApplicationShutdown()\* | Được gọi sau khi tất cả các kết nối được đóng |

#### Testing

* testing có nghĩa là kiểm tra code của chúng ta có đáp ứng kỳ vọng hay không . Ví dụ: một function có tên là Tổng sẽ trả về kết quả phép tính cộng một dãy số được người dùng nhập vào.
* Để viết được unit test ta có thể sử dụng jest. Jest là một trình chạy thử nghiệm JavaScript, nghĩa là một thư viện javascript để tạo, và để kiểm tra các thục tục hay hàm trong js. Jest là một package có sẵn trong NPM, bạn có thể cài đặt nó trong bất kỳ dự án JavaScript nào. Jest là một trong những trình chạy thử nghiệm phổ biến nhất hiện nay và được sử dụng nhiều trong các dự án
* Chi tiết cách sử dụng và cách tích hợp cần phải tìm hiểu thêm

### Technical

#### Database

* Nestjs cho phép bạn dễ dàng tích hợp với bất kỳ cơ sở dữ liệu SQL hoặc NoSQL nào. Bạn có một số tùy chọn có sẵn, tùy thuộc vào sở thích của bạn. Ở cấp độ chung nhất, việc kết nối Nest với cơ sở dữ liệu chỉ đơn giản là load driver Node.js thích hợp cho cơ sở dữ liệu, giống như cách bạn làm với Express hoặc Fastify
* Để thuận tiện, Nest cung cấp khả năng tích hợp chặt chẽ với TypeORM và Sequelize
* Cách kết nối thông qua 2 phương pháp này đã được giới thiệu ở trên

#### Mongo

* Nestjs cung cấp hai phương pháp để tích hợp với cơ sở dữ liệu MongoDB. Bạn có thể sử dụng module TypeORM tích hợp, module này có trình kết nối cho MongoDB hoặc sử dụng Mongoose, công cụ kết nối đến MongoDB phổ biến nhất.
* Chi tiết cách kết nối cần phải tìm hiểu thêm

#### Configuration

* Các ứng dụng thường chạy trong các môi trường khác nhau. Tùy thuộc vào môi trường, nên sử dụng các cài đặt cấu hình khác nhau. Ví dụ, thông thường môi trường local sẽ có những config cho môi trường local, dev sẽ có config cho môi trường dev, prodution sẽ có cấu hình cho môi trường production
* Các biến môi trường trong nodejs có thể được thấy thông qua method process.env. Trong các ứng dụng nodejs ta có thể sử dụng file .env sử dụng các cặp key-value đại diện cho một giá trị cụ thể ứng với từng môi trường khác nhau. Cụ thể trong nestjs cung cấp package ConfigModule có thể cấu hình .env cho từng môi trường khác nhau một cách thuận tiện và quản lý tốt hơn. Chi tiết cần phải tìm hiểu thêm

#### Validation

* Đây là cách tốt nhất là xác nhận tính đúng đắn của bất kỳ dữ liệu nào được gửi vào ứng dụng web. Để tự động xác thực các request đến, Nest cung cấp một số pipe có sẵn ngay khi phản hồi :
  + ValidationPipe
  + ParseIntPipe
  + ParseBoolPipe
  + ParseArrayPipe
  + ParseUUIDPipe
* ValidationPipe sử dụng gói package [class-validator](https://github.com/typestack/class-validator) và sử dụng decorator khi khai báo. Chi tiết các chức năng và cách sử dụng cần phải tìm hiểu thêm

#### Caching

* Là công nghệ đơn giản giúp gia tăng hiệu năng cho ứng dụng của bạn. Nó hoạt động như một kho lưu trữ dữ liệu tạm thời cung cấp khả năng truy cập dữ liệu hiệu suất cao.

#### Serialization

* Serialization là một quá trình xảy ra trước khi các đối tượng được trả về trong một phản hồi từ mạng. Đây là nơi thích hợp để cung cấp các quy tắc chuyển đổi và làm sạch dữ liệu sẽ được trả lại cho client. Ví dụ: dữ liệu nhạy cảm như mật khẩu phải luôn được loại trừ khỏi response. Hoặc, một số thuộc tính nhất định có thể yêu cầu chuyển đổi bổ sung, chẳng hạn như chỉ gửi một tập hợp con các thuộc tính của một thực thể. Thực hiện các phép biến đổi này theo cách thủ công có thể dễ xảy ra lỗi, đồng thời có thể khiến bạn không chắc chắn rằng tất cả các trường hợp đã được bảo vệ.
* Nest cung cấp một khả năng tích hợp để giúp đảm bảo rằng các hoạt động này có thể được thực hiện một cách đơn giản. ClassSerializerInterceptor sử dụng package [class-transformer](https://github.com/typestack/class-transformer) để cung cấp một cách khai báo và có thể mở rộng đối tượng biến đổi. Chi tiết cần phải tìm hiểu thêm

#### Task Schedule

Task Schedule cho phép chúng ta lập lịch mã (phương pháp / chức năng) tùy ý để thực thi vào một ngày / giờ cố định, theo khoảng thời gian định kỳ hoặc một lần sau một khoảng thời gian xác định. Trong thế giới Linux, điều này thường được xử lý bởi các gói như cron ở cấp hệ điều hành. Đối với ứng dụng Node.js, có một số gói mô phỏng chức năng giống như cron. Nest cung cấp gói @nestjs/schedule, tích hợp với gói Node.js node-cron phổ biến.

Cài đặt:

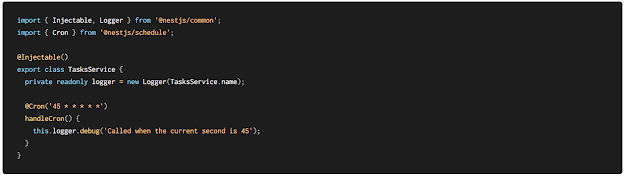
**$ npm install --save @nestjs/schedule**

**$ npm install --save-dev @types/cron**

Nhập ScheduleModule vào AppModule và chạy phương thức tĩnh forRoot() như bên dưới hình



Khai báo một công việc cron với trình trang trí @Cron () trước định nghĩa phương thức chứa mã sẽ được thực thi, như sau:



Trong ví dụ trên, Chúng ta đã chuyển 45 \* \* \* \* \* cho decorator



#### Queues

Queue là một mẫu thiết kế mạnh mẽ giúp chúng ta đối phó với các thách thức về quy mô và hiệu suất ứng dụng phổ biến. Ở ứng dụng nestjs, ta sử dụng @nestjs/bull

Bull là một thư viện Node triển khai một hệ thống queue nhanh chóng và mạnh mẽ dựa trên Redis. Mặc dù có thể triển khai queue trực tiếp chúng tag lệnh Redis, Bull là một trừu tượng / trình bao bọc trên Redis. Nó cung cấp một API xử lý tất cả các chi tiết cấp thấp và làm phong phú thêm chức năng cơ chúng ta của Redis để có thể dễ dàng xử lý các trường hợp sử dụng phức tạp hơn.

Nói một cách dễ hiểu hơn thì...

[Bull](https://github.com/OptimalBits/bull) là một trong số các thư viện hỗ trợ [message queue](https://www.cloudamqp.com/blog/2014-12-03-what-is-message-queuing.html) rất tốt trên NodeJS (bên cạnh [RabbitMQ](https://www.rabbitmq.com/tutorials/tutorial-three-javascript.html), [Kafka](https://kafka.apache.org/), [rsmq](https://github.com/smrchy/rsmq), [agenda](https://github.com/agenda/agenda),...), và dựa trên Redis để lưu dữ liệu.

Message queue là giải pháp để giải quyết các vấn đề trong công nghệ, thường là backend:

* Xử lý background.
* Gửi thông tin theo hàng đợi.
* Xử lý đồng bộ, concurrency.

Bull là thư viện có thể giúp cả 3 vấn đề trên, và chúng ta thân cũng có 3 thành phần chính:

* Producer: Thành phần tạo ra các công việc (job) và ném chúng vào queue (hàng đợi).
* Consumer: Thành phần nhận việc từ queue.
* Listener: Thành phần lắng nghe các sự kiện diễn ra trong hàng đợi (completed, failed, stalled).

Về cơ chúng ta, message queue là cơ chế đưa các thông tin vào dãy để xử lý dần, và mặc định với Bull thì hàng đợi là dạng FIFO (dữ liệu vào trước thì ra trước).

#### Cookies

Cookie HTTP là một phần dữ liệu nhỏ được trình duyệt của người dùng lưu trữ. Cookie được thiết kế để trở thành một cơ chế đáng tin cậy để các trang web ghi nhớ thông tin trạng thái. Khi người dùng truy cập lại trang web, cookie sẽ tự động được gửi kèm theo yêu cầu.

Chúng ta có thể viết nó vào trong main như sau:

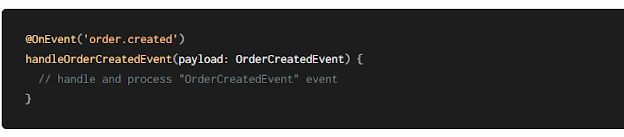


[Event Emitter](https://www.npmjs.com/package/@nestjs/event-emitter) package (@nestjs/event-emitter) cung cấp một triển khai quan sát đơn giản, cho phép chúng ta đăng ký và lắng nghe các sự kiện khác nhau xảy ra trong ứng dụng của chúng ta

Khi Emit 1 sự kiện:



Khi lắng nghe 1 Emit:



##### 

##### 

#### File upload

Để xử lý quá trình tải tệp lên, Nest cung cấp một mô-đun tích hợp dựa trên gói phần mềm trung gian nhiều tầng cho Express.

Để tải lên một tệp, chỉ cần kết nối bộ chặn FileInterceptor () với trình xử lý định tuyến và trích xuất tệp từ yêu cầu chúng tag cách sử dụng @UploadedFile ()



##### 

##### 

#### Streaming file

Có thể có những lúc chúng ta muốn gửi lại một tệp từ API REST của mình cho khách hàng. Để thực hiện việc này với Nest, thông thường chúng ta làm như sau:

****

##### 

##### 

#### Axios

Là gói ứng dụng khách HTTP phong phú được sử dụng rộng rãi. Nest bao bọc Axios và hiển thị nó thông qua tích hợp sẵn HttpModule. Và HttpModule exports HttpService class. trong đó hiển thị các phương thức dựa trên Axios để thực hiện các yêu cầu HTTP. Thư viện cũng chuyển đổi các phản hồi HTTP kết quả thành Observables.

#### Session

Các phiên HTTP cung cấp một cách để lưu trữ thông tin về người dùng qua nhiều yêu cầu, điều này đặc biệt hữu ích cho các ứng dụng MVC.

## Section 4

### Final project

Đề bài bài toán:

Hệ thống loyalty

- Admin

+ Quản lý danh sách tất cả các user (CRUD)

+ Quản lý danh sách tất cả các store (CRUD)

+ Phê duyệt cửa hàng mới khi đăng ký

- Store

+ Đăng ký (sử dụng OTP để verify), đăng nhập bằng email (phải sau khi được duyệt mới có thể đăng nhập được)

+ Quản lý danh sách user tại cửa hàng

+ Tích điểm cho user

+ Tạo danh sách các item dùng để đổi quà (tên, điểm đổi, ảnh, thời hạn, số lượng, mô tả)

- User

+ User đăng ký một tài khoản và có thể áp dụng đổi quà tại tất cả các của hàng của hệ thống

+ User đăng ký/đăng nhập bằng số điện thoại và gửi OTP để xác nhận

+ Khi user mua hàng tại cửa hàng, số điểm tích cho user được chia làm 2 dạng

\* Tích điểm cố định (Ví dụ: 100k được tích 100 điểm, 200k được tích 200 điểm)

\* Tích điểm theo phần trăm giá trị đơn hàng (Ví dụ: 100k được tích 10%, tối đa 5k điểm, 200k được tích 20% tối đa 10k điểm)

+ Số điểm tích của user sẽ được chia thành các cấp bậc rank(mặc định khi đăng ký sẽ là rank đồng)

\* Rank đồng(mặc định): Cứ mỗi 100k được tích 5 điểm hoặc cứ mỗi 100k được tích 10% tối đa 5 điểm

\* Rank bạc (khi tích được 2000 điểm sẽ lên rank): Cứ mỗi 100k được tích 10 điểm hoặc cứ mỗi 100k được tích 15% tối đa 10 điểm

\* Rank vàng (khi tích được 5000 điểm sẽ lên rank): Cứ mỗi 100k được tích 15 điểm hoặc cứ mỗi 100k được tích 20% tối đa 20 điểm

+ Đổi sản phẩm tại các cửa hàng(cần kiểm soát sô lượng đổi và sô lượng điểm trừ hay các điều kiện khác)