Tài liệu coding convention

1. Quy tắc đặt tên biến

1.1 Quy tắc chung

- Chỉ sử dụng các chữ cái và chữ số ASCII. Đặt tên mô tả càng rõ càng tốt, làm cho code dễ hiểu ngay lập tức đối với người đọc mới

  Ví dụ:

```
errorCount          // không viết tắt
dnsConnectionIndex  // Đọc là hiểu luôn Dns này để làm gì .
referrerUrl         // Url là của 1 referrer
customerId          // "Id" này được hiểu là Id của customer
```

- Không sử dụng các từ viết tắt hoặc không quen thuộc với người đọc:

  Ví dụ:

```
n                   // Ý nghĩa quá ngắn và viết tắt.
nErr                // Viết tắt
wgcConnections      // Người dùng đọc sẽ k hiểu wgc là gì
```

1.2 Quy tắc đặt tên packages

- Tên packages đều là lowerCamelCase

  Ví dụ:

```
// Nên
my.exampleCode.deepSpace

// Không nên
my.examplecode.deepspace
my.example_code.deep_space
```

1.3 Quy tắc đặt tên cho class, interface, enum

- Tên loại thường là danh từ hoặc cụm danh từ

  Ví dụ `Request, ImmutableList hoặc VisibilityMode`

- Tên interface thường phải là tính từ, có chữ `I` đằng trước để thể hiện là interface. Đối với enum nên có chữ `E` đằng trước để thể hiện là enum

  Ví dụ: `IReadable, IWritable, EUser, EType`

1.4 Quy tắc đặt tên cho methods

- Tên methods thường là động từ hoặc cụm động từ `sendMessage, getFoo, setFoo`

1.5 Quy tắc đặt tên cho constants

- Tất cả các chữ cái viết hoa, với các từ được phân tách bằng dấu gạch dưới: `CONSTANT_NAME`

1.6. Quy tắc đặt tên cho các field trong database

- Các field trong database sẽ đặt dạng `snake_case`, nhưng trong code khi sử dụng sequlize (typeORM) cần setup option để chuyển về dạng `camelCase`

  Ví dụ:

```
Trong database đặt tên là customer_id
@Table({
    underscored: true
})
Khi đó sẽ được chuyển thành customerId
```

2. Khai báo biến

- Khai báo tất cả các biến trong chương trình bằng let hoặc const, `không khai báo biến bằng var`. Sử dụng const sẽ là mặc định trừ khi biến đó cần được thay đổi giá trị.
- Mỗi lần khai báo chỉ được khai báo một biến

  Ví dụ:

```
const a = 1 // Nên
const a = 1, b = 2 // Không nên
var a = 1 // Không nên
```

- Các biến không được khai báo theo thói quen khi bắt đầu block, thay vào đó nên khai báo gần với block chúng được sử dụng lần đầu tiên để giảm thiểu phạm vi của biến

  Ví dụ:

```
const a = 1
function (a) {
    //Logic sử dụng biến a
}
```

- Loại bỏ việc khai báo kiểu dữ liệu của biến khi đã được định nghĩa khai báo quá rõ ràng

  Ví dụ:

```
// Nên
const a = 15

// Không nên
const x: boolean = true;
```

3. Quy tắc với source (file/folder/path)

3.1 Đặt tên file/folder:

- Tên file/folder phải là tất cả chữ thường
- Có thể dấu gạch dưới (_) hoặc dấu gạch ngang (-)
- Không dùng dấu câu, tiếng việt để đặt.
- Thực hiện theo quy ước mà dự án của bạn sử dụng. Phần mở rộng của tên file phải là .js/.ts.

  Ví dụ:

```
// nên
/booking/user-detail/details.js

// không nên
/booking/chi-tiết/DETAILS.js
```

3.2 Quy tắc với ES module

- Việc import các module khác có thể sử dụng đường dẫn tương đối hoặc đường dẫn tuyệt đối
- Nên sử dụng đường dẫn tương đối để không ảnh hưởng đến việc thay đổi vị trí các file, folder, ...

```
import {Symbol1} from 'path/from/root';
import {Symbol2} from '../parent/file';
import {Symbol3} from './sibling';
```

- Import không được ngắt dòng và giới hạn 80 cột.

```
import * as goog from '../closure/goog/goog.js';
```

- Không nhập cùng một file/folder nhiều lần.

```
// Không nên
import { short } from './long/path/to/a/file.js';
import { aLongNameThatBreaksAlignment } from './long/path/to/a/file.js';
// Nên?
```

- Áp dụng quy tắc đặt tên biến khi import modules

```
import * as fileOne from '../file-one.js';
import * as libString from './lib/string.js';
```

- Sử dụng exports có trong code. Có thể áp dụng export cho một khai báo hoặc sử dụng export { name };
- Không sử dụng default exports

```
// Không nên dùng
export default class Foo { ... } // BAD!
// Nên dùng
export class Foo { ... }
```

- Trong project, tất cả các file interface, enum, helper, ... khi được sử dụng ở nhiều chỗ cần có file index.ts để gom tất cả về một mối. Khi sử dụng chỉ import file index.ts này để sử dụng, tránh import nhiều file ở nhiều chỗ

  Ví dụ:

```
// File user.enum.ts
export enum EUserType {
    ADMIN = 'ADMIN',
    CLIENT = 'CLIENT',
}

// File index.ts
export * from './user.enum';

// Khi sử dụng chỉ import từ file index.ts ra
import { EUserType } from './../enums/index';
export interface IUser {
    id: string;
    name: string;
    userType: EUserType;
}
```

4. Coding Format

4.1 Dấu ngoặc nhọn

- Empty block có thể ngắn gọn. Ví dụ: `function doNothing() {}`
- Với các đoạn logic ngắn chỉ có 1 dòng có thể viết ngay trên dòng của chính nó.

  Ví dụ:

```
if (shortCondition()) foo();
```

4.2 Comments

- Với comment 1 dòng nên để dấu // với comment nhiều dòng nên để trong /* ... */

  Ví dụ:

```
/*
 * Đây là
 * được phép
 */

// Đây cũng
// là được phép

/* Đây cũng là được phép */
```

- Các biến hay function được comment như sau:

```
someFunction(obviousParam, /* shouldRender= */ true, /* name= */ 'hello');
```

5. Toán tử so sánh

- Sử dụng toán tử 3 dấu bằng (`===` và `!==`) khi thực hiện các phép so sánh 2 giá trị. Toán tử 2 dấu bằng sẽ ép kiểu và đôi khi dẫn đến lỗi khi xử lý logic

  Ví dụ

```
// Không nên
if (foo == 'bar' || baz != bam) {
    // Logic xử lý
}

// Nên
if (foo === 'bar' || baz !== bam) {
    // Logic xử lý
}
```

- Ngoại lệ duy nhất là khi chung ta so sánh với `null` bởi vì trong trường hợp đó nó sẽ xử lý cho cả `null` và `undefined`

  Ví dụ:

```
// Ngoại lệ: Nó sẽ handle cho trường hợp foo là null hoặc undefined.
if (foo == null) {
    // Logic xử lý
}
```

6. Dữ liệu trả về

- Khai báo interface với các biến dạng object
- Luôn trả về kiểu dữ liệu đối với các hàm, function

  Ví dụ:

```
//nên
interface IAnimal {
    sound: string;
    name: string;
}

const cat: IAnimal = {
    sound: 'meow',
};

// Không nên
const cat = {
    sound: 'meow',
};


//nên
interface IAnimal {
    sound: string;
    name: string;
}
function handlerLogic(): IAnimal  {

}

// Không nên
function handlerLogic() {

}
```

7. Về việc sử dụng object

- Không sử dụng các key có dấu nháy đơn (`''`)

  Ví dụ:

```
// nên
{
    width: 42,
    maxWidth: 43,
}

// không nên
{
    width: 42,
    'maxWidth': 43,
}
```

- Khai báo object Enum phải đảm bảo các giá trị không được thay đổi

  Ví dụ:

```
const Option = {
    FIRST_OPTION: 1,
    SECOND_OPTION: 2,
};
```

8. Về chuỗi ký tự

- Sử dụng dấu `'` (nháy đơn) thay vì dấu `"` (nháy kép) để tạo sự thống nhất, các chuỗi. Trong trường hợp chuỗi ký tự có chứa các tham số để gán vào thì có thể sử dụng dấu ` `` ` (cạnh phím ESC)
- Với các chuỗi ký tự dài không sử dụng dấu `\` để ngắt xuống dòng

  Ví dụ:

```
// không nên
const longString = 'This is a very long string that far exceeds the 80 \
    column limit. It unfortunately contains long stretches of spaces due \
    to how the continued lines are indented.';

// nên
const longString = 'This is a very long string that far exceeds the 80 ' +
'column limit. It does not contain long stretches of spaces since ' +
'the concatenated strings are cleaner.';

// nên
const longString = `This is a very long string that far exceeds the 80
column limit. It does not contain long stretches of spaces since
the concatenated strings are cleaner.`;
```

9. Về các cấu trúc điều khiển

9.1. Vòng lặp

- Luôn ưu tiên sử dụng vòng lặp `for - of` khi có thể

9.2 sử dụng switch - case

- Với mỗi một case khi kết thúc cần phải `return` giá trị hoặc phải sử dụng `break` cuối cùng
- Nên có `default` là hàm xử lý mặc định để tránh các trường hợp xử lý lỗi phía trên

  Ví dụ:

```
switch (input) {
case 1:
case 2:
    prepareOneOrTwo();
// sẽ bị nhảy qua         // không nên
case 3:
    handleOneTwoOrThree();
    break;                  // nên
default:                  // nên
    handleLargeNumber(input);
    return true
}
```

10. Exception

10.1. Khi trả lỗi thì dùng new

- Luôn luôn sử dụng new `Exception()` khi trả lỗi.

  Ví dụ:

```
throw new Exception("Lỗi message ở đây");
```

10.2. Chỉ throw lỗi

- Không throw `string` hay các giá trị khác. Chỉ được throw lỗi hoặc các `class` kế thừa của lỗi.

  Ví dụ:

```
// Không được throw như thế này
throw 'Lỗi message';

// Chỉ được throw lỗi
throw new Error('Lỗi message!');

// ... Hoặc là các class kế thừa từ class lỗi
class MyError extends Error {}
throw new MyError('Lỗi message custom!');
```

10.3. Catch lỗi và re-throw

- Trong `catch` khi xử lý logic lỗi thì luôn phải trả lại lỗi đó sau khi xử lý xong để không phá vỡ luồng xử lý lỗi của hệ thống.

  Ví dụ:

```
try {
    doSomething();
} catch (e: unknown) {
    // Khi xử lý lỗi xong phải re-throw lỗi
    assert(e, isInstanceOf(Error));
    displayError(e.message);
    throw new Eror(e)
}
```

- Type của `error` khi lấy ra để xử lý sẽ là `unknown` thay vì `any`

  Ví dụ:

```
try {
    doSomething();
} catch (e: unknown) {
    throw e;
}
```

11. Sử dụng spread operator

- Sử dụng spread operator `[...foo]` `{...bar}` để sao chép mảng và đối tượng

  Ví dụ:

```
const foo = {
    num: 1,
};

const foo2 = {
    ...foo,
    num: 5,
};

const foo3 = {
    num: 5,
    ...foo,
}

foo2.num === 5;
foo3.num === 1;
```

- Khi tạo một mảng hoặc object, chỉ spread các object không spread các giá trị khác kể cả `null` và `undefined`.

  Ví dụ:

```
// Không được sử dụng như này
const foo = {num: 7};
const bar = {num: 5, ...(shouldUseFoo && foo)}; // (shouldUseFoo && foo) có thể bằng undefined

// =>> Đổi sang như này
const foo = shouldUseFoo ? {num: 7} : {};
const bar = {num: 5, ...foo}; //foo sẽ không bao giờ spread undefined
```

12. Thêm block để kiểm soát được logic

- Khi viết code có nhưng đoạn logic sau câu `if`, `switch`, vòng lặp, không được bỏ đoạn `block {}`

  Ví dụ:

```
// Khi bỏ block {} đoạn code có thể làm rối đoạn logic của cả 1 function
if (x)
doSomethingWithALongMethodNameThatForcesANewLine(x);

for (let i = 0; i < x; i++) doSomethingWith(i);

// Luôn thêm block {} để đảm bảo logic được bọc trong 1 phạm vi nhất định
for (let i = 0; i < x; i++) {
    doSomethingWith(i);
}

if (x) {
    doSomethingWithALongMethodNameThatForcesANewLine(x);
}
```

- Ngoại lệ duy nhất nếu câu `if` ngắn mà được viết cùng 1 dòng

  Ví dụ

```
if (x) doFunctionInOneLine(x);
```

13. Try tường minh

- Chỉ đưa đoạn code logic có khả năng xảy ra lỗi vào đoạn `try catch`

  Ví dụ:

```
// Không nên - Quá nhiều logic trong 1 đoạn try - catch
try {
    const result = methodThatMayThrow();
    use(result);
} catch (error: unknown) {
    // xử lý lỗi
}

// Nên - Tách nhỏ từng phần
const result; // Khai báo biến
try {
    result = methodThatMayThrow(); // Logic chính
} catch (error: unknown) {
    // Xử lý lỗi
}
use(result); // Handle sau logic
```

- Ngoại lệ duy nhất là có thể bọc đoạn `try catch` cho cả 1 đoạn logic vòng lặp.

  Ví dụ:

```
try {
    while () {
        // Việc khởi tạo nhiều try - catch có thể ảnh hưởng đến vấn đề hiệu năng
    }
} catch (error: unknown) {
    // Xử lý lỗi
}
```

14. Viết function

14.1 Sử dụng arrow function cho callback

- Chỉ sử dụng arrow function khi trong callback

  Ví dụ:

```
// Không nên - Khai báo function với từ khoá function theo kiểu cũ
bar(function() { ... })

// Nên - Sử dụng arrow function
bar(() => { this.doSomething(); })
```

14.2 Viết function hợp lý đúng kiểu đúng chỗ

- Viết `function` rút gọn hoặc là viết `function` có `block` phù hợp với nội dung đoạn code

  Ví dụ:

```
// Hàm cao nhất ưu tiên khai báo theo dạng sử dụng keyword functionn
function someFunction() {

    const receipts = books.map((b: Book) => { // Sử dụng arrow function trong callback
        const receipt = payMoney(b.price);
        recordTransaction(receipt);
        return receipt;
    });

    // Có thể sử dụng function rút gọn nếu function chỉ chứa 1 biểu thức và trả ra kết quả
    const longThings = myValues.filter(v => v.length > 1000).map(v => String(v));

    // Khi sử dụng function rút gọn để tính toán biểu thức thì phải được gán vào 1 biến const.
    const computeTax = (amount: number) => amount * 0.12;
}
```

- Hàm cao nhất ưu tiên khai báo theo dạng sử dụng keyword `function`

  Ví dụ:

```
function someFunction() {
    // Xử lý logic
}
```

- Block arrow function trong `callback`

  Ví dụ:

```
const receipts = books.map((b: Book) => {
    // Xử lý logic
});
```

- Có thể sử dụng `function` rút gọn nếu `function` chỉ chứa 1 biểu thức hoặc hàm và trả ra kết quả

  Ví dụ:

```
const longThings = myValues.filter(v => v.length > 1000).map(v => String(v));
```

- Khi sử dụng `function` rút gọn để tính toán biểu thức hoặc hàm thì phải được gán vào 1 biến `const`.

  Ví dụ:

```
const computeTax = (amount: number) => amount * 0.12;
```

- Không viết 1 function rút gọn nếu không trả ra kết quả gì, khi đó bắt buộc phải thêm block để xử lý logic

  Ví dụ:

```
// Không nên: Sử dụng kiểu function cho các hàm return ra void
myPromise.then(v => console.log(v));

// Nên: Thêm 1 đoạn block ({...})
myPromise.then(v => {
    console.log(v);
});
```

- Dùng thêm block (`{...}`) để trong trường hợp muốn tách nhỏ đoạn code để dễ đọc

  Ví dụ:

```
// Không nên: Viết 1 function kiểu rút gọn nhưng rất dài
const transformed = [1, 2, 3].map(v => worthWrapping(acrossManyLines(someComplicatedExpr(v)));
);

// Nên: Tách nhỏ code để dễ đọc.
const transformed = [1, 2, 3].map(v => {
    const intermediate = someComplicatedExpr(v);
    const more = acrossManyLines(intermediate);
    return worthWrapping(more);
});
```

15. Quy tắc đặt tên API

15.1. Xác định tài nguyên API

- Trong REST API, việc đặt tên cho các nguồn dữ liệu chính mà chúng ta sử dụng được gọi là tài nguyên (hay còn gọi là `Resources`)
- Một `Resources` có thể được chia ra thành các `Singleton Resources` và `Collection Resources`

  Ví dụ:

```
- Có "customers" là một collection resources và "customer" là một singleton resources.
- Có thể xác định "customers" khi sử dụng URI là "/customers", và có thể xác định singleton resource "customer" trong URI là "/customers/{customerId}
```

- Một `Resources` cũng có thể chứa nhiều các `Sub Resources` và được thể hiện như sau

  Ví dụ:

```
- Có sub-collection resources "accounts" là một phần của "customers" và có thể được xác định khi sử dụng URI như sau: "/customers/{customerId}/accounts" (Được thể hiện trong ngữ cảnh là một chủ thể  "customer"
- Tương tự nếu muốn lấy 1 singleton resource "account" của "accounts" : /customers/{customerId}/accounts/{accountId}
```

15.2. Quy tắc đặt tên

- Sử dụng danh từ để đại diện cho các `Resources` thay cho một động từ vì danh từ thường có các thuộc tính mà động từ không có
- Để rõ ràng hơn, chia các loại `Resources` ra thành 4 nhóm: document, collection, store và controller. Cụ thể như sau:

  a. Document

  - Là dạng `Resources` chỉ các đối tượng độc lập, 1 bản ghi trong database, có thể xem nó như môt tài nguyên duy nhất, sử dụng danh từ số ít để định danh nhóm này

    Ví dụ:

```
http://api.example.com/user-management/users/{id}
http://api.example.com/user-management/users/admin
```

  b. Collection

  - Là các `Resources` được quản lý bởi server, là các resource dành cho bên phía admin, quản trị. Sử dụng các tên số nhiều với loại resources này
    Ví dụ:

```
http://api.example.com/device-management/managed-devices
http://api.example.com/user-management/users
http://api.example.com/user-management/users/{id}/accounts
```

  c. Store

  - Là các `Resources` được quản lý bởi client - tức là người sử dụng API đó. Client có toàn quyền CRUD với resource này. Cũng sử dụng các tên số nhiều với loại resources này

    Ví dụ:

```
http://api.example.com/song-management/users/{id}/playlists
```

  d. Controller

  - Loại `Resources` này như đại diện cho 1 hành động, 1 quá trình và có input-output, hoặc 1 giá trị. Với các loại này hơi đặc biệt, nên sử dụng động từ để dễ hình dung.

    Ví dụ:

```
http://api.example.com/song-management/users/{id}/playlists
http://api.example.com/api/send-email
```

- Sử dụng "/" để ngăn cách quan hệ trong resource và không sử dụng "/" ở cuối api

  Ví dụ:

```
// Nên
http://api.example.com/device-management

// Không nên
http://api.example.com/device-management/
```

- Sử dụng "-" để tăng khả năng liền mạch đối với các path dài. Không sử dụng "_" để thay thế hoặc viết liền

  Ví dụ:

```
// Nên
http://api.example.com/api/messenger-management/group-chat/{id}

// Không nên
http://api.example.com/api/messenger_management/group_chat/{id}
http://api.example.com/api/messengermanagement/groupchat/{id}
```

- Sử dụng hoàn toàn bằng chữ cái viết thường, không xen kẽ với các chữ viết hoa hoặc toàn bộ viết hoa

  Ví dụ:

```
// Nên
http://api.example.com/device-management

// Không nên
http://api.example.com/DEVICE-MANAGEMENT
```

- Không phân định CRUD trên URI, tất cả đều được xác định thông qua các method GET, POST,PUT, DELETE, ...

  Ví dụ:

```
// Nên
HTTP GET http://api.example.com/user-management/users -> get tất cả user
HTTP POST http://api.example.com/user-management/users -> tạo mới user

// Không nên
http://api.example.com/user-management/users/get
http://api.example.com/user-management/users/create
```

- Sử dụng query để filter các URI collection. Giả sử có API để lấy tất cả các User. Nhưng rồi lại có thêm 1 requirement như trả về các user theo từng trang với số lượng nhất định hoặc lọc ra các người dùng với quốc tịch là Việt Nam chẳng hạn. Thay vì viết API mới mất nhiều thời gian, nên sử dụng các API cũ với thêm các query truyền vào

  Ví dụ:

```
http://api.example.com/user-management/users?page=1&size=10
http://api.example.com/user-management/users?nationality=Vietnam
```