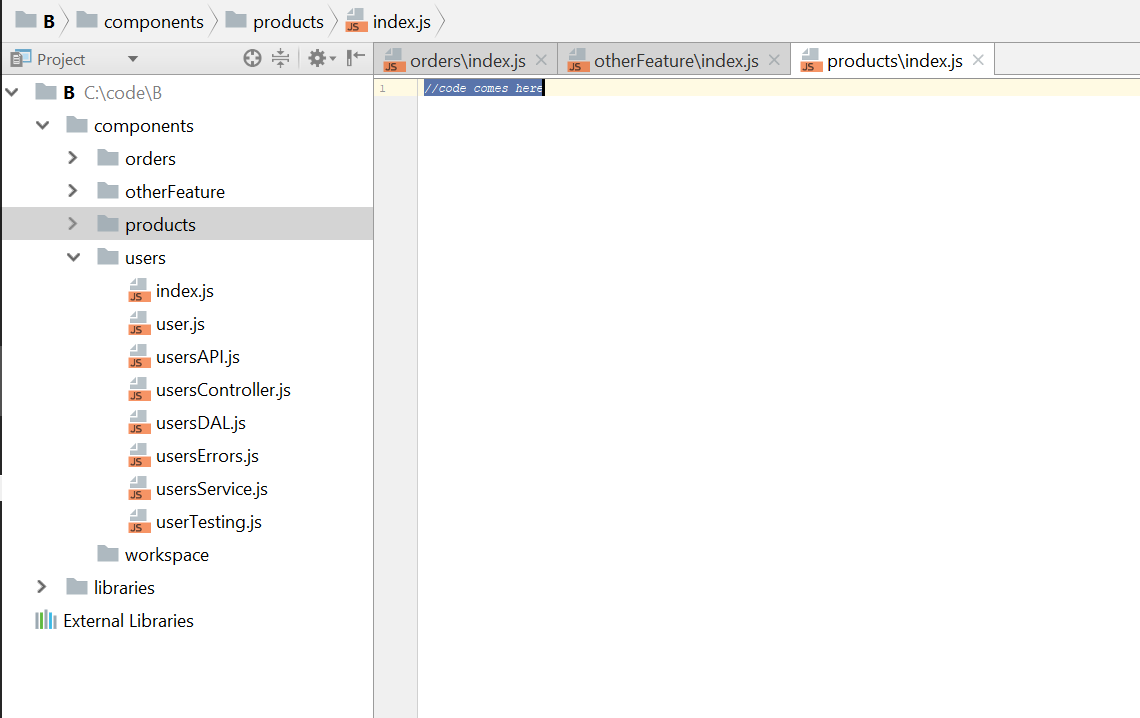
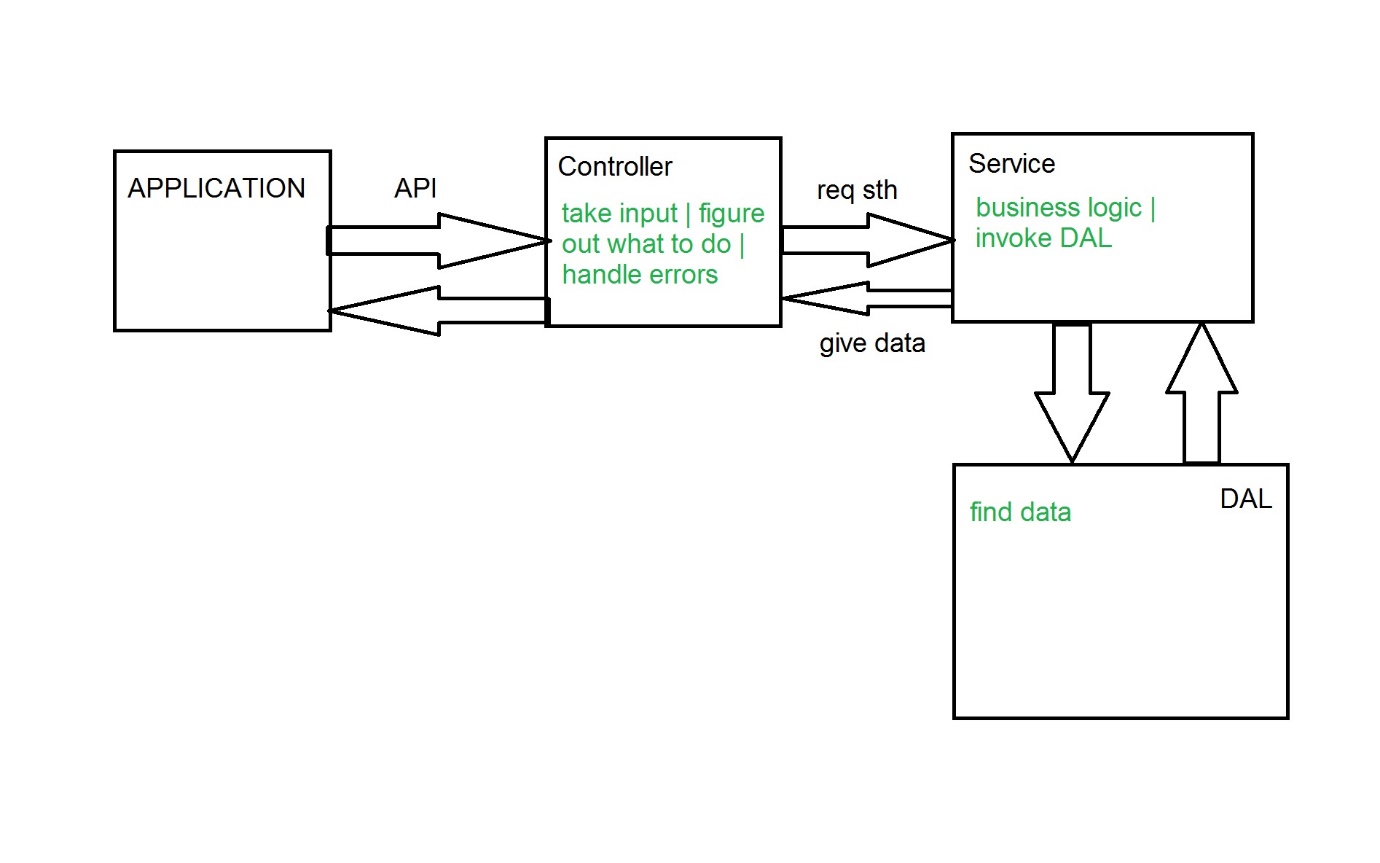
به نام خدا

Best practices

1- Structure





Explanation: differences between controller | service | DAL  
<https://github.com/i0natan/nodebestpractices/issues/314>

2- Error  
 2.1 use async await or promises for async error handling.  
 callback is a bridge to hell.

2.2 use throw new Error('error')

Or even better make your own error handling but using Error class.

2.3 Distinguish operational vs programmer errors

Operational errors (trusted errors) refer to situations where you understand what happened and the impact of it. (for example a query failed)

Programmer errors refer to cases where you have no idea why and sometimes where an error came from. it might be some code that tried to read an undefined value or DB connection pool that leaks memory

Logging operational errors is enough. But for programmer errors maybe there's nothing better you can do than to restart gracefully.

// marking an error object as operational

const myError = new Error("How can I add new product when no value provided?");

myError.isOperational = true;

// or if you're using some centralized error factory (see other examples at the bullet "Use only the built-in Error object")

class AppError {

constructor (commonType, description, isOperational) {

Error.call(this);

Error.captureStackTrace(this);

this.commonType = commonType;

this.description = description;

this.isOperational = isOperational;

}

};

throw new AppError(errorManagement.commonErrors.InvalidInput, "Describe here what happened", true);

2.4 Handle errors centrally, not within an Express middleware

Without one dedicated object for error handling, greater are the chances of important errors hiding under the radar due to improper handling

I think it is exact what I have in my codes.!!!

2.5 Document API errors using Swagger

Let your API callers know which errors might come in return so they can handle these thoughtfully without crashing.

STATUS CODES ARE REALLY IMPORTANT IN RETRUNS.

2.6 Exit the process gracefully when a stranger comes to town

When an unknown error occurs (a developer error, see best practice 2.3) - there is uncertainty about the application healthiness. A common practice suggests restarting the process carefully using a process management tool like Forever or PM2.

2.7 Use a mature logger to increase error visibility

Forget about console.log and use something like Winston, Bunyan or Log4J

2.8 Test error flows using your favorite test framework!!!!!!!!!!!!!!!!!!!!!!!!

؟؟؟؟؟؟؟؟؟؟؟؟؟؟؟؟؟؟؟؟؟؟؟؟؟؟؟؟؟؟؟؟؟؟؟؟؟؟؟؟؟؟؟؟؟؟؟؟؟؟؟؟؟؟؟؟؟؟؟؟؟؟

2.9 Discover errors and downtime using APM (application performance management) products

UpTimeRobot.com | AppDynamics.com

2.10 Catch unhandled promise rejections

Don't forget to write ". catch ".

**Code example: these errors will not get caught by any error handler (except unhandledRejection)**

DAL.getUserById(1).then((johnSnow) => {

// this error will just vanish

if(johnSnow.isAlive == false)

throw new Error('ahhhh');

});

If you forget you must have this:

### Code example: Catching unresolved and rejected promises

process.on('unhandledRejection', (reason, p) => {

// I just caught an unhandled promise rejection, since we already have fallback handler for unhandled errors (see below), let throw and let him handle that

throw reason;

});

process.on('uncaughtException', (error) => {

// I just received an error that was never handled, time to handle it and then decide whether a restart is needed

errorManagement.handler.handleError(error);

if (!errorManagement.handler.isTrustedError(error))

process.exit(1);

});

2.11 Fail fast, validate arguments using a dedicated library

Use validation libraries to avoid bugs from API callers or your admin panel.

Libraries: joi | validator | express-validator

### Code example: validating complex JSON input using ‘Joi’

var memberSchema = Joi.object().keys({

password: Joi.string().regex(/^[a-zA-Z0-9]{3,30}$/),

birthyear: Joi.number().integer().min(1900).max(2013),

email: Joi.string().email()

});

function addNewMember(newMember) {

// assertions come first

Joi.assert(newMember, memberSchema); //throws if validation fails

// other logic here

}

### Anti-pattern: no validation yields nasty bugs

// if the discount is positive let's then redirect the user to print his discount coupons

function redirectToPrintDiscount(httpResponse, member, discount) {

if (discount != 0) {

httpResponse.redirect(`/discountPrintView/${member.id}`);

}

}

redirectToPrintDiscount(httpResponse, someMember);

// forgot to pass the parameter discount, why the heck was the user redirected to the discount screen?

3 CODE STYLE PRACTICES

3.1 Use ESLint

Beautifier ESLint prettier

3.2 Node.js specific plugins

3.3 Start a Codeblock's curly braces on the same line.

// Do

function someFunction() {

// code block

}

// Avoid

function someFunction()

{

// code block

}

3.4 Don't forget semicolon

// Do

const count = 2;

(function doSomething() {

// do something amazing

}());

// Avoid — throws exception

const count = 2 // it tries to run 2(), but 2 is not a function

(function doSomething() {

// do something amazing

}())

3.5 Name your functions

Name all functions, including closures and callbacks. Avoid anonymous functions

3.6 Use naming conventions for variables, constants, functions and classes

Use **lowerCamelCase** when naming constants, variables and functions and **UpperCamelCase** (capital first letter as well) when naming classes. This will help you to easily distinguish between plain variables/functions, and classes that require instantiation. Use descriptive names, but try to keep them short

I use UpperCamelCase for models too.

3.7 Prefer const over let. Ditch the var

Using const means that once a variable is assigned, it cannot be reassigned. Preferring const will help you to not be tempted to use the same variable for different uses, and make your code clearer. If a variable needs to be reassigned, in a for loop, for example, use let to declare it. Another important aspect of let is that a variable declared using it is only available in the block scope in which it was defined. var is function scoped, not block scoped, and [shouldn't be used in ES6](https://hackernoon.com/why-you-shouldnt-use-var-anymore-f109a58b9b70) now that you have const and let at your disposal

3.8 require modules first, not inside functions

3.9 require modules by folder, opposed to the files directly

When developing a module/library in a folder, place an index.js file that exposes the module's internals so every consumer will pass through it. This serves as an 'interface' to your module and eases future changes without breaking the contract

**Otherwise:** Changing the internal structure of files or the signature may break the interface with clients

### Code example

// Do

module.exports.SMSProvider = require('./SMSProvider');

module.exports.SMSNumberResolver = require('./SMSNumberResolver');

// Avoid

module.exports.SMSProvider = require('./SMSProvider/SMSProvider.js');

module.exports.SMSNumberResolver = require('./SMSNumberResolver/SMSNumberResolver.js');

3.10 Use the === operator

### Code example

'' == '0' // false

0 == '' // true

0 == '0' // true

false == 'false' // false

false == '0' // true

false == undefined // false

false == null // false

null == undefined // true

' \t\r\n ' == 0 // true

All statements above will return false if used with ===

3.11 Use Async Await, avoid callbacks

3.12 Use arrow function expressions ( => )

4 TESTING AND OVERALL QUALITY PRACTICES

4.1 Start with API testing rather than unit testing!!(Learn more about API testing here : <https://www.soapui.org/learn/functional-testing/api-testing-101.html> ).