# [Node.js Style Guide](https://github.com/felixge/node-style-guide" \l "nodejs-style-guide)

## [Table of contents](https://github.com/felixge/node-style-guide" \l "table-of-contents)

### [Formatting](https://github.com/felixge/node-style-guide" \l "formatting)

* [2 Spaces for indentation](https://github.com/felixge/node-style-guide" \l "2-spaces-for-indentation)
* [Newlines](https://github.com/felixge/node-style-guide" \l "newlines)
* [No trailing whitespace](https://github.com/felixge/node-style-guide" \l "no-trailing-whitespace)
* [Use Semicolons](https://github.com/felixge/node-style-guide" \l "use-semicolons)
* [80 characters per line](https://github.com/felixge/node-style-guide" \l "80-characters-per-line)
* [Use single quotes](https://github.com/felixge/node-style-guide" \l "use-single-quotes)
* [Opening braces go on the same line](https://github.com/felixge/node-style-guide" \l "opening-braces-go-on-the-same-line)
* [Declare one variable per var statement](https://github.com/felixge/node-style-guide" \l "declare-one-variable-per-var-statement)

### [Naming Conventions](https://github.com/felixge/node-style-guide" \l "naming-conventions)

* [Use lowerCamelCase for variables, properties and function names](https://github.com/felixge/node-style-guide" \l "use-lowercamelcase-for-variables-properties-and-function-names)
* [Use UpperCamelCase for class names](https://github.com/felixge/node-style-guide" \l "use-uppercamelcase-for-class-names)
* [Use UPPERCASE for Constants](https://github.com/felixge/node-style-guide" \l "use-uppercase-for-constants)

### [Variables](https://github.com/felixge/node-style-guide" \l "variables)

* [Object / Array creation](https://github.com/felixge/node-style-guide" \l "object--array-creation)

### [Conditionals](https://github.com/felixge/node-style-guide" \l "conditionals)

* [Use the === operator](#_Use the === operator)
* [Use multi-line ternary operator](https://github.com/felixge/node-style-guide" \l "use-multi-line-ternary-operator)
* [Use descriptive conditions](https://github.com/felixge/node-style-guide" \l "use-descriptive-conditions)

### [Functions](https://github.com/felixge/node-style-guide" \l "functions)

* [Write small functions](https://github.com/felixge/node-style-guide" \l "write-small-functions)
* [Return early from functions](https://github.com/felixge/node-style-guide" \l "return-early-from-functions)
* [Name your closures](https://github.com/felixge/node-style-guide" \l "name-your-closures)
* [No nested closures](https://github.com/felixge/node-style-guide" \l "no-nested-closures)
* [Method chaining](https://github.com/felixge/node-style-guide" \l "method-chaining)

### [Comments](https://github.com/felixge/node-style-guide" \l "comments)

* [Use slashes for comments](https://github.com/felixge/node-style-guide" \l "use-slashes-for-comments)

### [Miscellaneous](https://github.com/felixge/node-style-guide" \l "miscellaneous)

* [Object.freeze, Object.preventExtensions, Object.seal, with, eval](https://github.com/felixge/node-style-guide" \l "objectfreeze-objectpreventextensions-objectseal-with-eval)
* [Requires At Top](https://github.com/felixge/node-style-guide" \l "requires-at-top)
* [Getters and setters](https://github.com/felixge/node-style-guide" \l "getters-and-setters)
* [Do not extend built-in prototypes](https://github.com/felixge/node-style-guide" \l "do-not-extend-built-in-prototypes)

## [Formatting](https://github.com/felixge/node-style-guide" \l "formatting-1)

You may want to use [editorconfig.org](http://editorconfig.org/) to enforce the formatting settings in your editor. Use the [Node.js Style Guide .editorconfig file](https://github.com/felixge/node-style-guide/blob/master/.editorconfig) to have indentation, news lines and white space behavior automatically set to the rules set up below.

<https://github.com/felixge/node-style-guide/blob/master/.editorconfig>

### [2 Spaces for indentation](https://github.com/felixge/node-style-guide" \l "2-spaces-for-indentation)

Use 2 spaces for indenting your code and swear an oath to never mix tabs and spaces - a special kind of hell is awaiting you otherwise.

### [Newlines](https://github.com/felixge/node-style-guide" \l "newlines)

Use UNIX-style newlines (\n), and a newline character as the last character of a file. Windows-style newlines (\r\n) are forbidden inside any repository.

### [No trailing whitespace](https://github.com/felixge/node-style-guide" \l "no-trailing-whitespace)

Just like you brush your teeth after every meal, you clean up any trailing whitespace in your JS files before committing. Otherwise the rotten smell of careless neglect will eventually drive away contributors and/or co-workers.

### [Use Semicolons](https://github.com/felixge/node-style-guide" \l "use-semicolons)

According to [scientific research](http://news.ycombinator.com/item?id=1547647), the usage of semicolons is a core value of our community. Consider the points of [the opposition](http://blog.izs.me/post/2353458699/an-open-letter-to-javascript-leaders-regarding), but be a traditionalist when it comes to abusing error correction mechanisms for cheap syntactic pleasures.

### [80 characters per line](https://github.com/felixge/node-style-guide" \l "80-characters-per-line)

Limit your lines to 80 characters. Yes, screens have gotten much bigger over the last few years, but your brain has not. Use the additional room for split screen, your editor supports that, right?

### [Use single quotes](https://github.com/felixge/node-style-guide" \l "use-single-quotes)

Use single quotes, unless you are writing JSON.

**Right**:

var foo = 'bar';

**Wrong**:

var foo = "bar";

### [Opening braces go on the same line](https://github.com/felixge/node-style-guide" \l "opening-braces-go-on-the-same-line)

Your opening braces go on the same line as the statement.

**Right**:

if (true) {

    console.log('winning');

}

**Wrong**:

if (true)

{

  console.log('losing');

}

Also, notice the use of white space before and after the condition statement.

### [Declare one variable per var statement](https://github.com/felixge/node-style-guide" \l "declare-one-variable-per-var-statement)

Declare one variable per var statement, it makes it easier to re-order the lines. However, ignore [Crockford](http://javascript.crockford.com/code.html) when it comes to declaring variables deeper inside a function, just put the declarations wherever they make sense.

**Right**:

var keys   = ['foo', 'bar'];

var values = [23, 42];

var object = {};

while (keys.length) {

  var key = keys.pop();

  object[key] = values.pop();

}

**Wrong**:

var keys = ['foo', 'bar'],

    values = [23, 42],

    object = {},

    key;

while (keys.length) {

  key = keys.pop();

  object[key] = values.pop();

}

## [Naming Conventions](https://github.com/felixge/node-style-guide" \l "naming-conventions-1)

### [Use lowerCamelCase for variables, properties and function names](https://github.com/felixge/node-style-guide" \l "use-lowercamelcase-for-variables-properties-and-function-names)

Variables, properties and function names should use lowerCamelCase. They should also be descriptive. Single character variables and uncommon abbreviations should generally be avoided.

**Right**:

var adminUser = db.query('SELECT \* FROM users ...');

**Wrong**:

var admin\_user = db.query('SELECT \* FROM users ...');

### [Use UpperCamelCase for class names](https://github.com/felixge/node-style-guide" \l "use-uppercamelcase-for-class-names)

Class names should be capitalized using UpperCamelCase.

**Right**:

function BankAccount() {

}

**Wrong**:

function bank\_Account() {

}

### [Use UPPERCASE for Constants](https://github.com/felixge/node-style-guide" \l "use-uppercase-for-constants)

Constants should be declared as regular variables or static class properties, using all uppercase letters.

**Right**:

var SECOND = 1 \* 1000;

function File() {

}

File.FULL\_PERMISSIONS = 0777;

**Wrong**:

const SECOND = 1 \* 1000;

function File() {

}

File.fullPermissions = 0777;

## [Variables](https://github.com/felixge/node-style-guide" \l "variables-1)

### [Object / Array creation](https://github.com/felixge/node-style-guide" \l "object--array-creation)

Use trailing commas and put short declarations on a single line. Only quote keys when your interpreter complains:

**Right**:

var a = ['hello', 'world'];

var b = {

  good: 'code',

  'is generally': 'pretty',

};

**Wrong**:

var a = [

    'hello', 'world'

  ];

  var b = {"good": 'code'

          , is generally: 'pretty'

          };

## [Conditionals](https://github.com/felixge/node-style-guide" \l "conditionals-1)

### [Use the === operator](https://github.com/felixge/node-style-guide" \l "use-the--operator)

Programming is not about remembering [stupid rules](https://developer.mozilla.org/en/JavaScript/Reference/Operators/Comparison_Operators). Use the triple equality operator as it will work just as expected.

**Right**:

var a = 0;

if (a !== '') {

  console.log('winning');

}

**Wrong**:

var a = 0;

if (a == '') {

  console.log('losing');

}

### [Use multi-line ternary operator](https://github.com/felixge/node-style-guide" \l "use-multi-line-ternary-operator)

The ternary operator should not be used on a single line. Split it up into multiple lines instead.

**Right**:

var foo = (a === b)

  ? 1

  : 2;

**Wrong**:

var foo = (a === b) ? 1 : 2;

### [Use descriptive conditions](https://github.com/felixge/node-style-guide" \l "use-descriptive-conditions)

Any non-trivial conditions should be assigned to a descriptively named variable or function:

**Right**:

var isValidPassword = password.length >= 4 && /^(?=.\*\d).{4,}$/.test(password);

if (isValidPassword) {

  console.log('winning');

}

**Wrong**:

if (password.length >= 4 && /^(?=.\*\d).{4,}$/.test(password)) {

    console.log('losing');

}

## [Functions](https://github.com/felixge/node-style-guide" \l "functions-1)

### [Write small functions](https://github.com/felixge/node-style-guide" \l "write-small-functions)

Keep your functions short. A good function fits on a slide that the people in the last row of a big room can comfortably read. So don't count on them having perfect vision and limit yourself to ~15 lines of code per function.

### [Return early from functions](https://github.com/felixge/node-style-guide" \l "return-early-from-functions)

To avoid deep nesting of if-statements, always return a function's value as early as possible.

**Right**:

function isPercentage(val) {

    if (val < 0) {

      return false;

    }

    if (val > 100) {

      return false;

    }

    return true;

}

**Wrong**:

function isPercentage(val) {

    if (val >= 0) {

      if (val < 100) {

        return true;

      } else {

        return false;

      }

    } else {

      return false;

    }

}

Or for this particular example it may also be fine to shorten things even further:

function isPercentage(val) {

    var isInRange = (val >= 0 && val <= 100);

    return isInRange;

}

### [Name your closures](https://github.com/felixge/node-style-guide" \l "name-your-closures)

Feel free to give your closures a name. It shows that you care about them, and will produce better stack traces, heap and cpu profiles.

**Right**:

req.on('end', function onEnd() {

    console.log('winning');

});

**Wrong**:

req.on('end', function () {

    console.log('losing');

});

### [No nested closures](https://github.com/felixge/node-style-guide" \l "no-nested-closures)

Use closures, but don't nest them. Otherwise your code will become a mess.

**Right**:

setTimeout(function () {

    client.connect(afterConnect);

}, 1000);

function afterConnect() {

    console.log('winning');

}

**Wrong**:

setTimeout(function () {

    client.connect(function () {

        console.log('losing');

    });

}, 1000);

### [Method chaining](https://github.com/felixge/node-style-guide" \l "method-chaining)

One method per line should be used if you want to chain methods.

You should also indent these methods so it's easier to tell they are part of the same chain.

**Right**:

User

  .findOne({ name: 'foo' })

  .populate('bar')

  .exec(function(err, user) {

    return true;

  });

**Wrong**:

User

.findOne({ name: 'foo' })

.populate('bar')

.exec(function(err, user) {

  return true;

});

User.findOne({ name: 'foo' })

  .populate('bar')

  .exec(function(err, user) {

    return true;

  });

User.findOne({ name: 'foo' }).populate('bar')

.exec(function(err, user) {

  return true;

});

User.findOne({ name: 'foo' }).populate('bar')

  .exec(function(err, user) {

    return true;

  });

## [Comments](https://github.com/felixge/node-style-guide" \l "comments-1)

### [Use slashes for comments](https://github.com/felixge/node-style-guide" \l "use-slashes-for-comments)

Use slashes for both single line and multi line comments. Try to write comments that explain higher level mechanisms or clarify difficult segments of your code. Don't use comments to restate trivial things.

**Right:**

// 'ID\_SOMETHING=VALUE' -> ['ID\_SOMETHING=VALUE', 'SOMETHING', 'VALUE']

var matches = item.match(/ID\_([^\n]+)=([^\n]+)/));

// This function has a nasty side effect where a failure to increment a

// redis counter used for statistics will cause an exception. This needs

// to be fixed in a later iteration.

function loadUser(id, cb) {

  // ...

}

var isSessionValid = (session.expires < Date.now());

if (isSessionValid) {

  // ...

}

**Wrong**:

// Execute a regex

var matches = item.match(/ID\_([^\n]+)=([^\n]+)/);

// Usage: loadUser(5, function() { ... })

function loadUser(id, cb) {

  // ...

}

// Check if the session is valid

var isSessionValid = (session.expires < Date.now());

// If the session is valid

if (isSessionValid) {

  // ...

}

## [Miscellaneous](https://github.com/felixge/node-style-guide" \l "miscellaneous-1)

### [Object.freeze, Object.preventExtensions, Object.seal, with, eval](https://github.com/felixge/node-style-guide" \l "objectfreeze-objectpreventextensions-objectseal-with-eval)

Crazy shit that you will probably never need. Stay away from it.

### [Requires At Top](https://github.com/felixge/node-style-guide" \l "requires-at-top)

Always put requires at top of file to clearly illustrate a file's dependencies. Besides giving an overview for others at a quick glance of dependencies and possible memory impact, it allows one to determine if they need a package.json file should they choose to use the file elsewhere.

### [Getters and setters](https://github.com/felixge/node-style-guide" \l "getters-and-setters)

Do not use setters, they cause more problems for people who try to use your software than they can solve.

Feel free to use getters that are free from [side effects](http://en.wikipedia.org/wiki/Side_effect_(computer_science)), like providing a length property for a collection class.

### [Do not extend built-in prototypes](https://github.com/felixge/node-style-guide" \l "do-not-extend-built-in-prototypes)

Do not extend the prototype of native JavaScript objects. Your future self will be forever grateful.

**Right**:

var a = [];

if (!a.length) {

  console.log('winning');

}

**Wrong**:

Array.prototype.empty = function() {

    return !this.length;

  }

  var a = [];

  if (a.empty()) {

    console.log('losing');

  }

### [Modules and Imports](https://github.com/felixge/node-style-guide" \l "do-not-extend-built-in-prototypes)

Use ES6 import/export syntax for module management. Avoid using require and module.exports if possible.

// Good

import fs from 'fs';

import { myFunction } from './myModule';

// Bad

const fs = require('fs');

const myFunction = require('./myModule');

### Error Handling

Always handle errors properly, whether by using try-catch blocks or passing errors to callback functions. Avoid using empty catch blocks.

try {

    // Code that may throw an error

} catch (error) {

    // Handle the error

}

### Async/Await

Use async/await for asynchronous code instead of callback hell or promises for improved readability.

async function fetchData() {

    try {

        const data = await fetchDataFromAPI();

        return data;

    } catch (error) {

        throw new Error('Failed to fetch data');

    }

}

### Project Structure Practices

### Structure your solution by components:

The worst large applications pitfall is maintaining a huge code base with hundreds of dependencies – such a monolith slows down developers as they try to incorporate new features.

Instead, partition your code into components, each gets its own folder or a dedicated codebase, and ensure that each unit is kept small and simple.

The ultimate solution is to develop small software: divide the whole stack into self-contained components that don’t share files with others, each constitutes very few files (e.g. API, service, data access, test, etc.) so that it’s very easy to reason about it.

### Layer your components

Each component should contain ‘layers’ – a dedicated object for the web, logic, and data access code. This not only draws a clean separation of concerns but also significantly eases mocking and testing the system.

### Wrap common utilities as npm packages

In a large app that constitutes a large codebase, cross-cutting-concern utilities like logger, encryption and alike, should be wrapped by your own code and exposed as private npm packages.

This allows sharing them among multiple codebases and projects.

### Separate Express ‘app’ and ‘server’

Avoid the habit of defining the entire Express app in a single huge file – separate your ‘Express’ definition to at least two files: the API declaration (app.js) and the networking concerns (WWW).

For even better structure, locate your API declaration within components

### Use environment aware, secure and hierarchical config

A perfect and flawless configuration setup should ensure (a) keys can be read from file AND from environment variable (b) secrets are kept outside committed code (c) config is hierarchical for easier findability.

There are a few packages that can help tick most of those boxes like [rc](https://www.npmjs.com/package/rc), [nconf](https://www.npmjs.com/package/nconf) and [config](https://www.npmjs.com/package/config).

### Document API errors using Swagger or GraphQL

Let your API callers know which errors might come in return so they can handle these thoughtfully without crashing. For RESTful APIs, this is usually done with documentation frameworks like Swagger.

### Use a mature logger to increase error visibility

A set of mature logging tools like [Winston](https://www.npmjs.com/package/winston), [Bunyan](https://github.com/trentm/node-bunyan), [Log4js](http://stritti.github.io/log4js/) or [Pino](https://github.com/pinojs/pino), will speed-up error discovery and understanding. So forget about console.log.

### Constantly inspect for vulnerable dependencies

Even the most reputable dependencies such as Express have known vulnerabilities.

This can get easily tamed using community and commercial tools such as [npm audit](https://docs.npmjs.com/cli/audit" \t "https://www.perfomatix.com/nodejs-coding-standards-and-best-practices/_blank) and [snyk.io](https://snyk.io/" \t "https://www.perfomatix.com/nodejs-coding-standards-and-best-practices/_blank) that can be invoked from your CI on every build.

## **Security Best Practices**

### Embrace linter security rules

Make use of security-related linter plugins such as [eslint-plugin-security](https://www.npmjs.com/package/eslint-plugin-security" \t "https://www.perfomatix.com/nodejs-coding-standards-and-best-practices/_blank) to catch security vulnerabilities and issues as early as possible, preferably while they’re being coded.

This can help to catch security weaknesses like using eval, invoking a child process or importing a module with a string literal (e.g. user input).

### Extract secrets from config files or use packages to encrypt them

Never store plain-text secrets in configuration files or source code.

Instead, make use of secret-management systems like Vault products, Kubernetes/Docker Secrets, or using environment variables.

As a last resort, secrets stored in source control must be encrypted and managed (rolling keys, expiring, auditing, etc). Make use of pre-commit/push hooks to prevent committing secrets accidentally.

### Adjust the HTTP response headers for enhanced security

Your application should be using secure headers to prevent attackers from using common attacks like cross-site scripting (XSS), clickjacking and other malicious attacks. These can be configured easily using modules like [helmet](https://www.npmjs.com/package/helmet" \t "https://www.perfomatix.com/nodejs-coding-standards-and-best-practices/_blank).

### Avoid using the Node.js crypto library for handling passwords, use Bcrypt

Passwords or secrets (API keys) should be stored using a secure hash + salt function like bcrypt, that should be a preferred choice over its JavaScript implementation due to performance and security reasons.

### Prevent brute-force attacks against authorization

A simple and powerful technique is to limit authorization attempts using two metrics:

1. The first is a number of consecutive failed attempts by the same user unique ID/name and IP address.
2. The second is a number of failed attempts from an IP address over some long period of time. For example, block an IP address if it makes 100 failed attempts in one day.

**Reference Links:**

<https://github.com/felixge/node-style-guide#2-spaces-for-indentation>

<https://www.perfomatix.com/nodejs-coding-standards-and-best-practices/>

<https://www.tatvasoft.com/blog/node-js-best-practices/>