Node.js Style Guide

This is a guide for writing consistent and aesthetically pleasing node.js code. It is inspired by what is popular within the community, and flavored with some personal opinions.

There is a .jshintrc which enforces these rules as closely as possible. You can either use that and adjust it, or use this script to make your own.

Table of contents

- No nested closures
- Method chaining

Comments

- Use slashes for comments

Miscellaneous

- Object.freeze, Object.preventExtensions, Object.seal, with, eval
- Requires At Top
- Getters and setters
- Do not extend built-in prototypes

Formatting

You may want to use editorconfig.org to enforce the formatting settings in your editor. Use the Node.js Style Guide .editorconfig file to have indentation, newslines and whitespace behavior automatically set to the rules set up below.

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

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

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

According to scientific research, the usage of semicolons is a core value of our community. Consider the points of the opposition, but be a traditionalist when it comes to abusing error correction mechanisms for cheap syntactic pleasures.

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

Use single quotes, unless you are writing JSON.

Right:

```
var foo = 'bar';
```

Wrong:

```
var foo = "bar";
```

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 whitespace before and after the condition statement.

Declare one variable per var statement

Declare one variable per var statement, it makes it easier to re-order the lines. However, ignore Crockford 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

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

Class names should be capitalized using `UpperCamelCase`.

Right:

```
function BankAccount() {
}
```

Wrong:

```
function bank_Account() {
}
```

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

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

Use the === operator

Programming is not about remembering stupid rules. 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

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

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

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

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

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

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

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

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

Object.freeze, Object.preventExtensions, Object.seal, with, eval

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

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

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, like providing a length property for a

collection class.

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');
}
```