Felix's Node.js Style Guide

There is no official document that governs the style of node.js applications. This guide is my opinionated attempt to bring you a good set of instructions that will allow you to create beautiful and consistent software.

This guide assumes that you are only targeting node.js. If your code also needs to run in the browser or other environments, please ignore some of it.

Please also note that node.js, as well as various packages for it, have their own slightly different styles. So if you're interested in contributing to those, play by their rules.

Tabs vs Spaces

Let's start with the religious problems first. Our benevolent dictator has chosen 2 space indention for the node core, so you would do well to follow his choice.

Semicolons

There are rebellious forces that try to steal your semicolons from you. But make no mistake, our traditional culture is still well and truly alive. So follow the community, and use those semicolons!

Editors

You can use any editor. However, having support for JS syntax highlighting and executing the currently open file with node.js will come in very handy. While vim may not help you to impress the ladies, it will please our BDFL and your grandpa will also approve.

I'm typing this document in Notes on my iPad, but that's because I'm on a beach in Thailand. It's likely that your own work environment will impact your choice of editor as well.

Trailing whitespace

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

Line length

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

Quotes

Use single quotes, unless you are writing JSON.

Right:

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

Wrong:

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

Braces

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.

Variable declarations

Declare one variable per var statement, it makes it easier to re-order the lines. Ignore Crockford on this, and put those declarations wherever they make sense.

Right:

```
var keys = ['foo', 'bar'];
var values = [23, 42];

var object = {};
while (items.length) {
  var key = keys.pop();
  object[key] = values.pop();
}
```

Wrong:

```
var keys = ['foo', 'bar'],
    values = [23, 42],
    object = {},
    key;

while (items.length) {
  key = keys.pop();
  object[key] = values.pop();
}
```

Variable and property names

Variables and properties should use lower camel case capitalization. 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 = d.query('SELECT * FROM users ...');
```

Class names

Class names should be capitalized using upper camel case.

Right:

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

Wrong:

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

Constants

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

Node.js / V8 actually supports mozilla's const extension, but unfortunately that cannot be applied to class members, nor is it part of any ECMA standard.

Right:

```
var SECOND = 1 * 1000;

function File() {
}
File.FULL_PERMISSIONS = 0777;
```

Wrong:

```
const SECOND = 1 * 1000;

function File() {
}
File.fullPermissions = 0777;
```

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'
        };
```

Equality 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');
}
```

Extending prototypes

Do not extend the prototypes of any objects, especially native ones. There is a special place in hell waiting for you if you don't obey this rule.

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');
}
```

Conditions

Any non-trivial conditions should be assigned to a descriptive variable:

Right:

```
var isAuthorized = (user.isAdmin() || user.isModerator());
if (isAuthorized) {
  console.log('winning');
}
```

Wrong:

```
if (user.isAdmin() || user.isModerator()) {
  console.log('losing');
}
```

Function length

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 ~10 lines of code per function.

Return statements

To avoid deep nesting of if-statements, always return a functions 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;
}
```

Named closures

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

Right:

```
req.on('end', function onEnd() {
  console.log('winning');
});
```

Wrong:

```
req.on('end', function() {
  console.log('losing');
});
```

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);
```

Callbacks

Since node is all about non-blocking I/O, functions generally return their results using callbacks. The convention used by the node core is to reserve the first parameter of any callback for an optional error object.

You should use the same approach for your own callbacks.

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

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

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.

EventEmitters

Node.js ships with a simple EventEmitter class that can be included from the 'events' module:

```
var EventEmitter = require('events').EventEmitter;
```

When creating complex classes, it is common to inherit from this EventEmitter class to emit events. This is basically a simple implementation of the Observer pattern.

However, I strongly recommend that you never listen to the events of your own class from within it. It isn't natural for an object to observe itself. It often leads to undesirable exposure to implementation details, and makes your code more difficult to follow.

Inheritance / Object oriented programming

Inheritance and object oriented programming are subjects by themselves. If you're interested in following this popular programming model, please read my Object oriented programming guide.