19 things I learnt reading the NodeJS docs – Hacker Noon

I'd like to think I know Node pretty well. I haven't written a web site that *doesn't* use it for about 3 years now. But I've never actually sat down and read the docs.

As long-time readers will know, I am on a journey of writing out every interface, prop, method, function, data type, etc related to web development, so that I can fill in the gaps of what I don't know. The Node docs were my last stop, having finished HTML, DOM, the Web APIs, CSS, SVG, and EcmaScript.

This held the most unknown gems for me, so I thought I'd share them in this little listicle. I will present them in descending order of appeal. As I do with my outfits when I meet someone new.

The querystring module as a general purpose parser

Let's say you've got data from some weirdo database that gives you an array of key/value pairs in the form `name:Sophie;shape:fox;condition:new`. Naturally you think this might be nice as a JavaScript object. So you create a blank object, then you create an array by splitting the string on `;`, loop over it, for each item, split it again on `:` and add the first item in that array as a prop of your object setting the second as the value.

Right?

No! You use `querystring`

```
const weirdoString = `name:Sophie;shape:fox;condition:new`;
const result = querystring.parse(weirdoString, `;`, `:`);
```

```
// result:
// {
//   name: `Sophie`,
//   shape: `fox`,
//   condition: `new`,
// };
```

The difference between nextTick and setImmediate

As with so many things, remembering the difference between these two is easy if you imagine they had different names.

`process.nextTick()` should be `process.sendThisToTheStartOfTheQueue()`.

`setImmediate()` should be called `sendThisToTheEndOfTheQueue()`.

(Unrelated: I always thought that in React, `props` should be called `stuffThatShouldStayTheSameIfTheUserRefreshes` and `state` should be called `stuffThatShouldBeForgottenIfTheUserRefreshes`. The fact that they're the same length is just a bonus.)

Server.listen takes an object

I'm a fan of passing a single 'options' parameter rather than five different parameters that are unnamed and must be in a particular order. As it turns out, you can do this when setting up a server to listen for requests.

```
require(`http`)
  .createServer()
  .listen({
    port: 8080,
    host: `localhost`,
  })
  .on(`request`, (req, res) => {
    res.end(`Hello World!`);
  });
```

This is a sneaky one, because it's not actually listed in the docs for `http.Server`, you will only find it in the `net.Server` documentation (which `http.Server` inherits from).

Relative paths

The path you pass to the `fs` module methods can be relative. It's relative to `process.cwd()`. This is probably something that most people already knew, but I always thought it had to be a full path.

```
const fs = require(`fs`);
const path = require(`path`);
```

```
// why have I always done this...
fs.readFile(path.join(__dirname, `myFile.txt`), (err, data) => {
  // do something
});
```

```
// when I could just do this?
fs.readFile(`./path/to/myFile.txt`, (err, data) => {
  // do something
});
```

Path parsing

One of the many things I've unnecessarily fiddled with regexes for is getting filenames and extensions from paths, when all I needed to do was:

```
myFilePath = `/someDir/someFile.json`;
path.parse(myFilePath).base === `someFile.json`; // true
path.parse(myFilePath).name === `someFile`; // true
path.parse(myFilePath).ext === `.json`; // true
```

Logging with colors

I'm going to pretend I didn't already know that `console.dir(obj, {colors: true})` will print the object with props/values as different colors, making them much easier to read.

You can tell setInterval() to sit in the corner

Let's say you use `setInterval()` to do a database cleanup once a day. By default, Node's event loop won't exit while there is a `setInterval()` pending. If you want to let Node sleep (I have no idea what the benefits of this are) you can do this.

```
const dailyCleanup = setInterval(() => {
  cleanup();
}, 1000 * 60 * 60 * 24);
```

```
dailyCleanup.unref();
```

Careful though, if you have nothing else pending (e.g. no http server listening), Node will exit.

Using the signal constants

If you enjoy killing, you may have done this before:

```
process.kill(process.pid, `SIGTERM`);
```

Nothing wrong with that. If no bug involving a typo had ever existed in the history of computer programming. But since that second parameter takes a string *or* the equivalent int, you can use the more robust:

```
process.kill(process.pid, os.constants.signals.SIGTERM);
```

IP address validation

There’s a built in IP address validator. I have written out a regex to do exactly this more than once. Stupid David.

`require(`net`).isIP(`10.0.0.1`)` will return `4`.

`require(`net`).isIP(`cats`)` will return `0`.

Because cats aren’t an IP address.

If you haven’t noticed, I’m going through a phase of using only backticks for strings. It’s growing on me but I’m aware it looks odd so I’m mentioning it and quite frankly wondering what the point of mentioning it is and am now concerned with how I’m going to wrap up this sentence, I shou

os.EOL

Have you ever hard-coded an end-of-line character?

Egad!

Just for you, there is `os.EOL` (which is `\r\n` on Windows and `\n` elsewhere). Switching to os.EOL will ensure your code behaves consistently across operating systems.

Edit: I didn’t give this enough thought. Andrew Meyer and Damon Gant have both pointed out in the comments that this can be troublesome. You must assume that any given file could use CRLF(`\r\n`) or LF (`\n`) and you simply don’t know.

If you have an open source project and want to enforce a certain line feed style, there is an eslint rule that can partially help with this. Doesn’t help if git then messes with it though.

There is still a use for `os.EOL` though. For example when writing out log files that won’t be transported to other OSes. In this case it ensures that the logs files are displayed correctly (e.g. when opened in notepad on a Windows server).

```
const fs = require(`fs`);
```

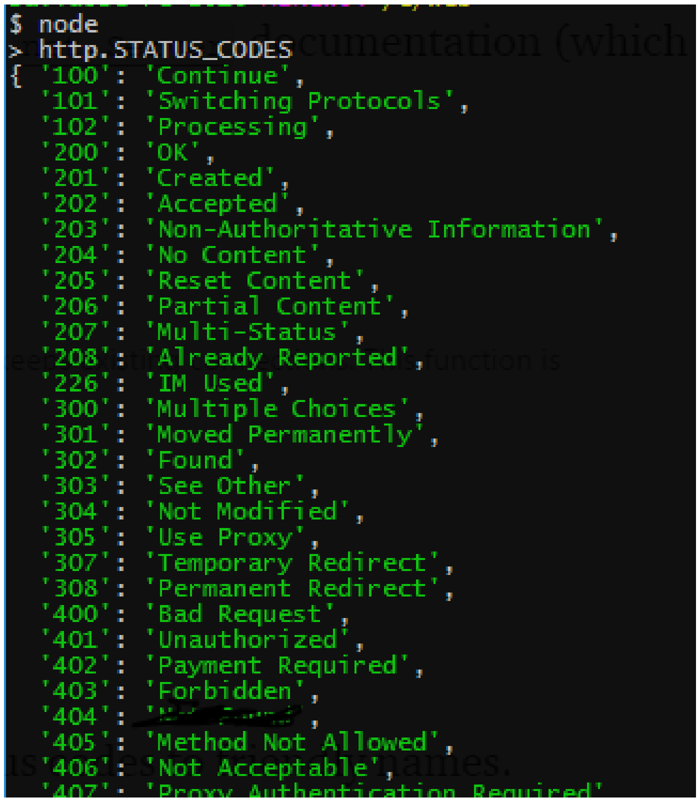
```
// hard-coded CRLF
```

```
fs.readFile(`./myFile.txt`, `utf8`, (err, data) => {
  data.split(`\r\n`).forEach(line => {
    // do something
  });
});
```

```
// based on OS
const os = require(`os`);
fs.readFile(`./myFile.txt`, `utf8`, (err, data) => {
  data.split(os.EOL).forEach(line => {
    // do something
  });
});
```

Status code lookup

There is a lookup of common HTTP status codes and their friendly names. `http.STATUS_CODES` is the object of which I speak, where each key is a status code, and the matching value is the human readable description.



```
$ node
> http.STATUS_CODES
{ '100': 'Continue',
  '101': 'Switching Protocols',
  '102': 'Processing',
  '200': 'OK',
  '201': 'Created',
  '202': 'Accepted',
  '203': 'Non-Authoritative Information',
  '204': 'No Content',
  '205': 'Reset Content',
  '206': 'Partial Content',
  '207': 'Multi-Status',
  '208': 'Already Reported',
  '226': 'IM Used',
  '300': 'Multiple Choices',
  '301': 'Moved Permanently',
  '302': 'Found',
  '303': 'See Other',
  '304': 'Not Modified',
  '305': 'Use Proxy',
  '307': 'Temporary Redirect',
  '308': 'Permanent Redirect',
  '400': 'Bad Request',
  '401': 'Unauthorized',
  '402': 'Payment Required',
  '403': 'Forbidden',
  '404': [redacted],
  '405': 'Method Not Allowed',
  '406': 'Not Acceptable',
  '407': 'Proxy Authentication Required',
```

So you can do this:

```
someResponse.code === 301; // true
require(`http`).STATUS_CODES[someResponse.code] === `Moved Permanently`; // true
```

Preventing unnecessary crashes

I always thought it was a bit ridiculous that the below would actually stop a server:

```
const jsonData = getDataFromSomeApi(); // But oh no, bad data!
const data = JSON.parse(jsonData); // Loud crashing noise.
```

To prevent silly things like this from ruining your day, you can put `process.on(`uncaughtException`, console.error);` right at the top of your Node app.

Of course, I'm a sane person so I use PM2 and wrap everything in `try...catch` if I'm being paid, but for personal projects…

Warning, this is not best practice, and in a large complex app is probably a bad idea. I'll let you decide if you want to trust some dude's blog post or the official docs.

Just this once()

In addition to `on()`, there is a `once()` method for all EventEmitters. I'm quite sure I'm the last person on earth to learn this. So … that's everyone then.

```
server.once(`request`, (req, res) => res.end(`No more from me.`));
```

Custom console

You can create your own console with `new console.Console(standardOut, errorOut)` and pass in your own output streams.

Why? I have no idea. Maybe you want to create a console that outputs to a file, or a socket, or a third thing.

DNS lookup

A little birdy told me that Node doesn't cache DNS lookups. So if you're hitting a URL again and again you're wasting valuable milliseconds. In which case you could get the domain yourself with `dns.lookup()` and cache it. Or here's one someone else created earlier.

```
dns.lookup(`www.myApi.com`, 4, (err, address) => {
  cacheThisForLater(address);
});
```

The `fs` module is a minefield of OS quirks

If, like me, your style of writing code is read-the-absolute-minimum-from-the-docs-then-fiddle-till-it-works, then you're probably going to run into trouble with the `fs` module. Node does a great job of ironing out the differences between operating systems, but there's only so much they can do, and a number of OS differences poke through the ocean of code like the jagged protrusions of a reef. A reef that has a minefield in it. You're a boat.

Unfortunately, these differences aren't all "Window vs The Others", so we can't simply invoke the "pffft, no one uses Windows" defence. (I wrote a whole big rant about the anti-Windows sentiment in web development but deleted it because it got so preachy even I was rolling my eyes at myself.)

Here is a spattering of things from the `fs` module that could bite you in your downstairs area:

- The `mode` property of the object returned by `fs.stats()` will be different on Windows and other operating systems (on Windows they may not match the file mode constants such as `fs.constants.S_IRWXU`).
- `fs.lchmod()` is only available on macOS.
- Calling `fs.symlink()` with the `type` parameter is only supported on Windows.
- The `recursive` option that can be passed to `fs.watch()` works on macOS and Windows only.
- The `fs.watch()` callback will only receive a filename on Linux and Windows.
- Using `fs.open()` on a directory with the flag `a+` will work on FreeBSD and Windows, but fail on macOS and Linux.
- A `position` parameter passed to `fs.write()` will be ignored on Linux when the file is opened in append mode (the kernel will ignore the position and append to the end of the file).

(I'm so hip I'm already calling it macOS when OS X has only been in the grave for 49 days.)

The net module is twice as fast as http

Reading the docs, I learnt that the `net` module is a thing. And that it underpins the `http` module. Which got me thinking, if I wanted to do server-to-server communication (as it turns out, I do) should I just be using the `net` module?

Networking folk will find it difficult to believe that I couldn't intuit the answer, but as a web developer that has fallen ass-first into the server world, I know HTTP and not much else. All this TCP, socket, stream malarkey is a bit like Japanese rap to me. Which is to say, I don't really get it, but I'm intrigued.

To play and compare, I set up a couple of servers (I trust you're listening to Japanese rap right now) and fired some requests at them. End result, `http.Server` handles ~3,400 requests per second, `net.Server` handles ~5,500 requests per second.

It's simpler too.

This is the code, if you're interested. If you're not interested then I'm sorry I'm making you scroll so much.

REPL tricks

1. If you're in the REPL (that is, you've typed `node` and hit enter in your terminal) you can type `.load someFile.js` and it will load that file in (for example you might load a file that sets a bunch of constants).
2. You can set the environment variable `NODE_REPL_HISTORY=""` to disable writing the repl history to a file. I also learnt (was reminded at least) that the REPL history is written to `~/.node_repl_history`, if you'd like a trip down memory lane.
3. `_` is a variable that holds the result of the last evaluated expression. Mildly handy!
4. When the REPL starts, all modules are loaded for you. So, for example, you can just type `os.arch()` so see what your architecture is. You don't need to do ``require(`os`).arch();``. (Edit: OK to be precise, modules are loaded on demand.)