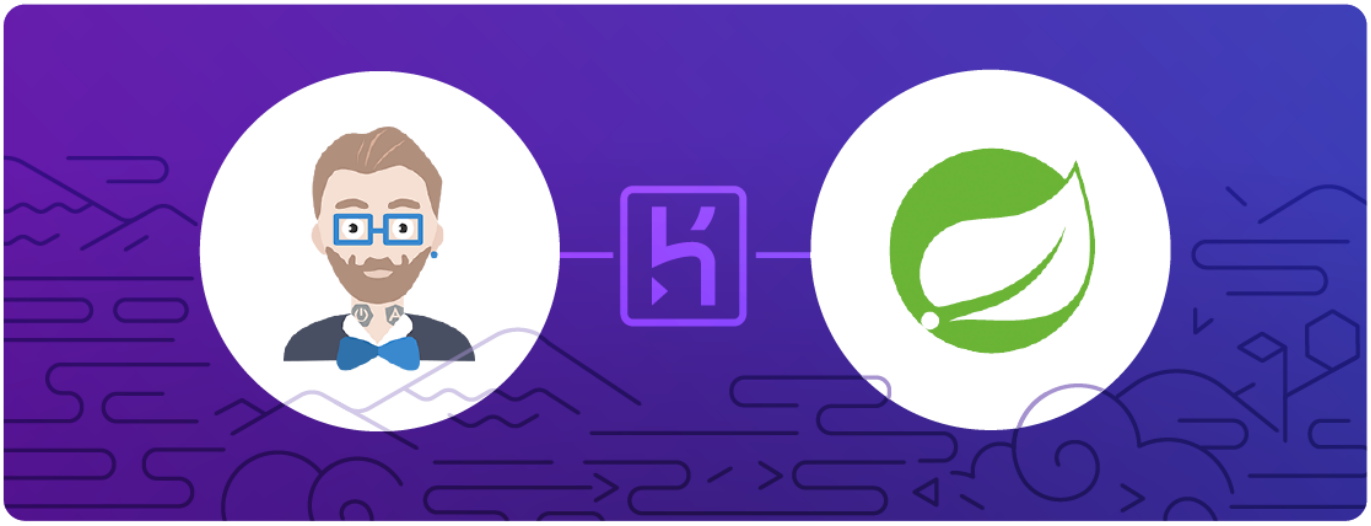

10 Habits of a Happy Node Hacker (2016)

November 10, 2015 by Hunter Loftis

At the tail end of 2015, JavaScript developers have a glut of tools at our disposal. The last time we looked into this, the modern JS landscape was just emerging. Today, it's easy to get lost in our huge ecosystem, so successful teams follow guidelines to make the most of their time and keep their projects healthy.

Here are ten habits for happy Node.js hackers as we enter 2016. They're specifically for *app developers*, rather than module authors, since those groups have different goals and constraints:

1. Start every new project with npm init

Npm's `init` command will scaffold out a valid package.json for your project, inferring common properties from the working directory.

```
$ mkdir my-awesome-app
$ cd my-awesome-app
$ npm init --yes
```

I'm lazy, so I run it with the `--yes` flag and then open package.json to make changes. The first thing you should do is specify an 'engines' key with your current version of node (`node -v`):

```
"engines": {
  "node": "4.2.1"
}
```

2. Use a smart .npmrc

By default, npm doesn't save installed dependencies to package.json (and you should always track your dependencies!).

If you use the `--save` flag to auto-update package.json, npm installs the packages with a leading carat (`^`), putting your modules at risk of drifting to different versions. This is fine for module development, but not good for apps, where you want to keep consistent dependencies between all your environments.

One solution is installing packages like this:

```
$ npm install foobar --save --save-exact
```

Even better, you can set these options in `~/.npmrc` to update your defaults:

```
$ npm config set save=true
$ npm config set save-exact=true
$ cat ~/.npmrc
```

Now, `npm install foobar` will automatically add `foobar` to package.json and your dependencies won't drift between installs!

If you prefer to keep flexible dependencies in package.json, but still need to lock down dependencies for production, you can alternatively build npm's shrinkwrap into your workflow. This takes a little more effort, but has the added benefit of preserving exact versions of nested dependencies.

3. Hop on the ES6 train

Node 4+ packs an updated V8 engine with several useful ES6 features. Don't be intimidated by some of the more complex stuff, you can learn it as you go. There are plenty of simple improvements for immediate gratification:

```
let user = users.find(u => u.id === ID);

console.log(`Hello, ${ user.name }!`);
```

4. Stick with lowercase

Some languages encourage filenames that match class names, like `MyClass` and 'MyClass.js'. Don't do that in node. Instead, use lowercase files:

```
let MyClass = require('my-class');
```

Node.js is the rare example of a Linux-centric tool with great cross-platform support. While OSX and Windows will treat 'myclass.js' and 'MyClass.js' equivalently, Linux won't. To write code that's portable between platforms, you'll need to exactly match `require` statements, *including capitalization*.

The easy way to get this right is to just stick with lowercase filenames for everything, eg 'my-class.js'.

5. Cluster your app

Since the node runtime is limited to a single CPU core and about 1.5 GB of memory, deploying a non-clustered node app on a large server is a huge waste of resources.

To take advantage of multiple cores and memory beyond 1.5 GB, bake Cluster support into your app. Even if you're only running a single process on small hardware today, Cluster gives you easy flexibility for the future.

Testing is the best way to determine the ideal number of clustered processes for your app, but it's good to start with the reasonable defaults offered by your platform, with a simple fallback, eg:

```
const CONCURRENCY = process.env.WEB_CONCURRENCY || 1;
```

Choose a Cluster abstraction to avoid reinventing the wheel of process management. If you'd like separate master and worker files, you can try forky. If you prefer a single entrypoint file and function, take a look at throng.

6. Be environmentally aware

Don't litter your project with environment-specific config files! Instead, take advantage of *environment variables*.

First, install node-foreman:

```
$ npm install --save --save-exact foreman
```

Next, create a Procfile to specify your app's process types:

```
web: bin/web
worker: bin/worker
```

Now you can start your app with the `nf` binary:

```
"scripts": {
  "start": "nf start"
}
```

To provide a local development environment, create a .gitignore'd `.env` file, which will be loaded by node-foreman:

```
DATABASE_URL='postgres://localhost/foobar'
HTTP_TIMEOUT=10000
```

Now, a single command (`npm start`) will spin up both a `web` process and a `worker` process in that environment. And, when you deploy your project, it will automatically adapt to the variables on its new host.

This is simpler and more flexible than 'config/abby-dev.js', 'config/brian-dev.js', 'config/qa1.js', 'config/qa2.js', 'config/prod.js', etc.

7. Avoid garbage

Node (V8) uses a lazy and greedy garbage collector. With its default limit of about 1.5 GB, it sometimes waits until it absolutely has to before reclaiming unused memory. If your memory usage is increasing, it might not be a leak - but rather node's usual lazy behavior.

To gain more control over your app's garbage collector, you can provide flags to V8 in your `Procfile`:

```
web: node --optimize_for_size --max_old_space_size=920 --gc_interval=100 server.js
```

This is especially important if your app is running in an environment with less than 1.5 GB of available memory. For example, if you'd like to tailor node to a 512 MB container, try:

```
web: node --optimize_for_size --max_old_space_size=460 --gc_interval=100 server.js
```

8. Hook things up

Npm's lifecycle scripts make great hooks for automation. If you need to run something before building your app, you can use the `preinstall` script. Need to build assets with grunt, gulp, browserify, or webpack? Do it in a `postinstall` script.

In package.json:

```
"scripts": {
  "postinstall": "bower install && grunt build",
  "start": "nf start"
}
```

You can also use environment variables to control these scripts:

```
"postinstall": "if $BUILD_ASSETS; then npm run build-assets; fi",
"build-assets": "bower install && grunt build"
```

If your scripts start getting out of control, move them to files:

```
"postinstall": "scripts/postinstall.sh"
```

Scripts in package.json automatically have `./node_modules/.bin` added to their `PATH`, so you can execute binaries like `bower` or `webpack` directly.

9. Only git the important bits

Most apps are composed of both necessary files and generated files. When using a source control system like git, you should avoid tracking anything that's generated.

For example, your node app probably has a `node_modules` directory for dependencies, which you should keep out of git. As long as each dependency is listed in package.json, anyone can create a working local copy of your app - including `node_modules` - by running `npm install`.

Tracking generated files leads to unnecessary noise and bloat in your git history. Worse, since some dependencies are native and must be compiled, checking them in makes your app less portable because you'll be providing builds from just a single, and possibly incorrect, environment.

For the same reason, you shouldn't check in `bower_components` or the compiled assets from grunt builds.

If you've accidentally checked in `node_modules` before, that's okay. You can remove it like this:

```
$ echo 'node_modules' >> .gitignore
$ git rm -r --cached node_modules
$ git commit -am 'ignore node_modules'
```

I also ignore npm's logs so they don't clutter my code:

```
$ echo 'npm-debug.log' >> .gitignore
$ git commit -am 'ignore npm-debug'
```

By ignoring these unnecessary files, your repositories will be smaller, your commits will be simpler, and you'll avoid merge conflicts in the generated directories.

10. Simplify

Tech predictions are famously inaccurate, but I'll make one here for the upcoming year. I predict that 2016 will be the year of *simplification* in JavaScript.

A growing group of developers are simplifying their architectures already. Instead of monolithic MVCs with big frameworks, they're building apps with static frontends, which can be served over CDN, with a Node.js API for dynamic data.

We're also beginning to see the drag that complex build systems put on our projects. The leading edge of developers are simplifying their builds - for instance, by using a 'vanilla' build without bower, gulp, or grunt.

Finally, we'll simplify our code in 2016. Sometimes, this will come from removing features, as with Douglas Crockford's "The Better Parts." Other times, this will come from adding features - like my favorite callback replacement, async-await. Async-await isn't yet available in Node, but you can use it today via the awesome BabelJS project.

Instead of seeing how many tools and frameworks you can cram in at once, try to simplify your work!

What are your habits?

I try to follow these habits in all of my projects. Whether you're new to node or a server-side JS veteran, I'm sure you've developed tricks of your own. We'd love to hear them! Share your habits by tweeting with the #node_habits hashtag.

Happy hacking!

Tags: es6, es7, npm, javascript, node

- Elements Marketplace
- Pricing

Resources

- Documentation
- Blog
- Get Started

About

- About Us
- What is Heroku
- Our Customers
- Careers
- Partners

Support

- Help
- Status
- Critical Apps
- Contact

Subscribe to our monthly newsletter

do not fill this in | Your email address | Go

- RSS Feed
- Twitter
- Facebook
- Github
- LinkedIn

Heroku is a company

- heroku.com
- Terms of Service
- Privacy
- Cookies