NODE.JS AT SCALE

Understanding the Node.js module system and using npm

From the Engineers of trace by RisingStack

Table of contents

CHAPTER ONE: NPM BEST PRACTICES

In the first chapter of the Node.js at Scale series, you will learn how npm can help you during the full lifecycle of your application - from starting a new project through development and deployment.

Get ready to learn tips and tricks that can save you a lot of time on a daily basis.

Know your npm

`npm install` is the most common way of using the npm cli - but it has a lot more to offer!

Before diving into the topics, let's see some commands that help you to know what version of npm you are running, or what commands are available.

npm versions

To get the version of the npm cli you are actively using, you can do the following:

```
$ npm --version
2.13.2
```

npm can return a lot more than just its own version - it can return the version of the current package, the Node.js version you are using and OpenSSL or V8 versions:

```
$ npm version
{ bleak: '1.0.4',
  npm: '2.15.0',
  ares: '1.10.1-DEV',
  http_parser: '2.5.2',
  icu: '56.1',
  modules: '46',
  node: '4.4.2',
  openssl: '1.0.2g',
  uv: '1.8.0',
  v8: '4.5.103.35',
  zlib: '1.2.8' }
```

npm help

As most cli toolkits, npm has a great built-in help functionality as well. Description and synopsis are always available. These are essentially man-pages.

```
NAME
       npm-test - Test a package

SYNOPSIS
           npm test [-- <args>]

           aliases: t, tst

DESCRIPTION
       This runs a package's "test" script, if
one was provided.

       To run tests as a condition of
installation, set the npat config to true.
```

1. START NEW PROJECTS WITH NPM INIT

When starting a new project, `npm init` can help you a lot by interactively creating a `package.json` file. This will prompt questions on the project's name or description for example. However, there is a quicker solution!

```
$ npm init --yes
```

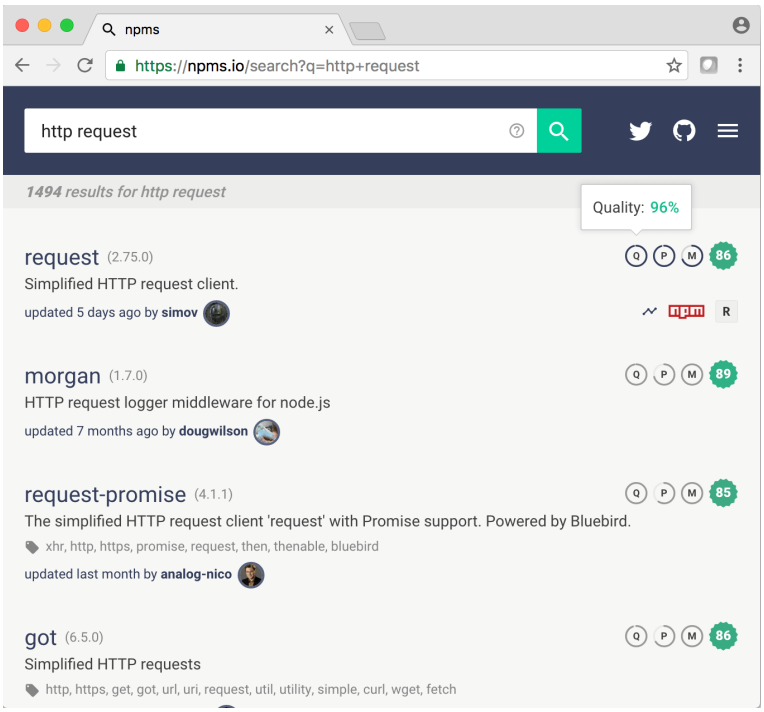
If you use `npm init --yes`, it won't prompt for anything, just create a `package.json` with your defaults. To set these defaults, you can use the following commands:

```
npm config set init.author.name YOUR_NAME
npm config set init.author.email YOUR_EMAIL
```

2. FINDING NPM PACKAGES

Finding the right packages can be quite challenging, since there are hundreds of thousands of modules you can choose from. We know this from experience, and developers participating in our latest Node.js survey also told us that selecting the right npm package is frustrating. Let's try to pick a module that helps us sending HTTP requests!

One website that makes the task a lot easier is npms.io. It shows metrics like **quality**, **popularity** and **maintenance**. These are calculated based on whether a module has outdated dependencies, does it have linters configured, is it covered with tests or when the most recent commit was made.



3. INVESTIGATE NPM PACKAGES

Once we picked our module (which will be the `request` module in our example), we should take a look at the documentation, and check out the open issues to get a better picture of what we are going to require into our application. Don't forget that the more npm packages you use, the higher the risk of having a vulnerable or malicious one. If you'd like to know more on npm-related security risks, read our related guideline.

If you'd like to open the homepage of the module from the cli you can do:

```
$ npm home request
```

To check open issues or the publicly available roadmap (if there's any), you can try this:

```
$ npm bugs request
```

Alternatively, if you'd just like to check a module's git repository, type this:

```
$ npm repo request
```

4. SAVING DEPENDENCIES

Once you found the package you want to include in your project, you have to install and save it. The most common way of doing that is by using `npm install request`.

If you'd like to take that one step forward and automatically add it to your package.json file, you can do:

```
$ npm install request --save
```

npm will save your dependencies with the `^` prefix by default. It means that during the next `npm install` the latest module without a major version bump will be installed. To change this behaviour, you can:

```
$ npm config set save-prefix='~'
```

In case you'd like to save the exact version, you can try:

```
$ npm config set save-exact true
```

5. LOCK DOWN DEPENDENCIES

Even if you save modules with exact version numbers as shown in the previous section, you should be aware that most npm module authors don't. It's totally fine, they do it to get patches and features automatically.

The situation can easily become problematic for production deployments:

It's possible to have different versions locally then on production, if in the meantime someone just released a new version. The problem will arise, when this new version has some bug which will affect your production system.

To solve this issue, you may want to use `npm shrinkwrap`. It will generate an `npm-shrinkwrap.json` that contains not just the exact versions of the modules installed on your machine, but also the version of its dependencies, and so on. Once you have this file in place, `npm install` will use it to reproduce the same dependency tree.

6. CHECK FOR OUTDATED DEPENDENCIES

To check for outdated dependencies, npm comes with a built-in tool method the `npm outdated` command. You have to run in the project's directory which you'd like to check.

```
$ npm outdated
conventional-changelog    0.5.3   0.5.3   1.1.0  @risingstack/docker-node
eslint-config-standard    4.4.0   4.4.0   6.0.1  @risingstack/docker-node
eslint-plugin-standard    1.3.1   1.3.1   2.0.0  @risingstack/docker-node
rimraf                    2.5.1   2.5.1   2.5.4  @risingstack/docker-node
```

Once you maintain more projects, it can become an overwhelming task to keep all your dependencies up to date in each of your projects. To automate this task, you can use Greenkeeper which will automatically send pull requests to your repositories once a dependency is updated.

7. NO DEVDEPENENDENCIES IN PRODUCTION

Development dependencies are called development dependencies for a reason - you don't have to install them in production. It makes your deployment artifacts smaller and more secure, as you will have less modules in production which can have security problems.

To install production dependencies only, run this:

```
$ npm install --production
```

Alternatively, you can set the `NODE_ENV` environment variable to production:

```
$ NODE_ENV=production npm install
```

8. SECURE YOUR PROJECTS AND TOKENS

In case of using npm with a logged in user, your npm token will be placed in the `.npmrc` file. As a lot of developers store dotfiles on GitHub, sometimes these tokens get published by accident. Currently, there are thousands of results when searching for the `.npmrc` file on GitHub, with a huge percentage containing tokens.

If you have dotfiles in your repositories, double check that your credentials are not pushed!

Another source of possible security issues are the files which are published to npm by accident. By default npm respects the `.gitignore` file, and files matching those rules won't be published. However, if you add an `.npmignore` file, it will override the content of `.gitignore` - so they won't be merged.

9. DEVELOPING PACKAGES

When developing packages locally, you usually want to try them out with one of your projects before publish to npm. This is where `npm link` comes to the rescue.

What `npm link` does is that it creates a symlink in the global folder that links to the package where the `npm link` was executed.

You can run `npm link package-name` from another location, to create a symbolic link from the globally installed `package-name` to the `/node_modules` directory of the current folder.

Let's see it in action!

```
# create a symlink to the global folder
/projects/request $ npm link

# link request to the current node_modules
/projects/my-server $ npm link request

# after running this project, the require('request')
# will include the module from projects/request
```

NEXT UP: SEMVER AND MODULE PUBLISHING

The next chapter of the Node.js at Scale series will be a SemVer deep dive with how to publish modules.

CHAPTER TWO:
SEMVER AND MODULE PUBLISHING

In the second chapter of this book, you are going to learn how to expand the npm registry with your own modules. We'll also going to explain how versioning works.

NPM MODULE PUBLISHING

When writing Node.js apps, there are so many things on npm that can help us being more productive. We don't have to deal with low-level things like padding a string from the left because there are already existing modules that are (eventually) available on the npm registry.

Where do these modules come from?

The modules are stored in a huge registry which is powered by a CouchDB instance.

The official public npm registry is at https://registry.npmjs.org/. It is powered by a CouchDB database, which has a public mirror at https://skimdb.npmjs.com/registry. The code for the couchapp is available at https://github.com/npm/npm-registry-couchapp.

How do modules make it to the registry?

People like you write them for themselves or for their co-workers and they share the code with their fellow JavaScript developers.

When should I consider publishing?

* If you want to share code between projects,
* if you think that others might run into the very same problem and you'd like to help them,
* if you have a bit (or even more) code that you think you can make use of later.

CREATING A MODULE

First let's create a module: `npm init -y` should take care of it, as you've learned in the previous chapter.

```
{
  "name": "npm-publishing",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "repository": {
    "type": "git",
    "url": "git+https://github.com/author/modulename"
  },
  "bugs": {
    "url": "https://github.com/caolan/async/issues"
  },
  "license": "ISC"
}
```

Let's break this down really quick. **These fields in your `package.json` are mandatory when you're building a module for others to use.**

First, you should give your module a distinct `name` because it has to be unique in the npm registry. Make sure it does not collide with any trademarks out there! `main` describes which file will be returned when your users do a `require('modulename')`. You can leave it as default or set it to any file in your project, but make sure you actually point it to a valid filename.

`keywords` should also be included because npm is going to index your package based on those fields and people will be able to find your module if they search those keywords in npm's search, or in any third party npm search site.

`author`, well obviously that's going to be you, but if anyone helps you develop your project be so kind to include them too! Also, it's very important to include where can people contact you.

In the `repository` field, you can see where the code is hosted and the `bugs` section tells you where can you file bugs if you find one in the package. To quickly jump to the bug report site you can use `npm bug modulename`.

1. Licensing

Solid license and licenses adoption helps Node adoption by large companies. Code is a valuable resource, and sharing it has it's own costs.

Licensing is a really hard, but this site can help you pick one that fits your needs.

Generally when people publish modules to npm they use the MIT license.

The MIT License is a permissive free software license originating at the Massachusetts Institute of Technology (MIT). As a permissive license, it puts only very limited restriction on reuse and has therefore an excellent license compatibility.

2. Semantic Versioning

Versioning is so important that it deserves its own section.

Most of the modules in the npm registry follow the specification called semantic versioning.

Semantic versioning describes the version of a software as 3 numbers separated by "."-s. It describes how this version number has to change when changes are made to the software itself. Given a version number **MAJOR.MINOR.PATCH**, increment the:

* **MAJOR** version when you make incompatible API changes,
* **MINOR** version when you add functionality in a backwards compatible manner, and
* **PATCH** version when you make backwards-compatible bug fixes.

Additional labels for the pre-release and the build metadata are available as extensions to the **MAJOR.MINOR.PATCH** format.

These numbers are for machines, not for humans! Don't assume that people will be discouraged from using your libraries when you often change the major version.

You have to start versioning at 1.0!

Most people think that doing changes while the software is still in "beta" phase should not respect the semantic versioning. They are wrong! It is really important to communicate **breaking changes** to your users even in beta phase. Always think about your users who want to experiment with your project.

3. Documentation

Having a proper documentation is imperative if you'd like to share your code with others. Putting a `README.md` file in your project's root folder is usually enough, and if you publish it to the registry npm will generate a site like this one. It's all done automatically and it helps other people when they try to use your code.

Before publishing, make sure you have all documentation in place and up to date.

4. Keeping secret files out of your package

Using a specific file called `.npmignore` will keep your secret or private files from publishing. Use that to your advantage, add files to `.npmignore` that you wish to not upload.

If you use `.gitignore` npm will use that too by default. Like git, npm looks for `.npmignore` and `.gitignore` files in all subdirectories of your package, not only in the root directory.

5. Encouraging contributions

When you open up your code to the public, you should consider adding some guidelines for them on how to contribute. Make sure they know how to help you dealing with software bugs and adding new features to your module.

There are a few of these available, but in general you should consider using github's issue and pull-request templates.

NPM PUBLISH

Now you understand everything that's necessary to publish your first module. To do so, you can type: `npm publish` and the npm-cli will upload the code to the registry.

Congratulations, your module is now public on the npm registry!

You can visit `www.npmjs.com/package/yourpackagename` for the public URL.

If you published something public to npm, it's going to stay there forever. There is little you can do to make it non-discoverable. Once it hits the public registry, **every other replica** that's connected to it will copy all the data. **Be careful when publishing.**

What if you published something that you didn't mean to?

We're human. We make mistakes, but what can be done now? Since the recent leftpad scandal, npm changed its unpublish policy.

If there is no package on the registry that depends on your package, then you're fine to unpublish it, but remember all the replicas will copy all the data - so someone somewhere will always be able to get it. If it contained any secrets, make sure you change them after the act, and remember to add them to the `.npmignore` file for the next publish.

Private Scoped Packages

If you don't want or you're not allowed to publish code to a public registry (for any corporate reasons), npm allows organizations to open an organization account so that they can push to the registry without being public. This way you can share private code between you and your co-workers.

Further read on how to set it up:
https://docs.npmjs.com/misc/scope

npm enterprise

If you'd like to further tighten your security by running a registry by yourself, you can do that pretty easily. npm has an on-premise version that can be run behind corporate firewalls. Read more about setting up npm enterprise.

LET'S BUILD SOMETHING!

Now that you know all these things, go and build something. If you're up for a little bragging, make sure you tweet us (@risingstack) the name of the package this book helped you to build!

In the next chapter, you're going to learn about how the module system, CommonJS and `require` works!

CHAPTER THREE:
HOW THE MODULE SYSTEM, COMMONJS & REQUIRE WORKS

In the third chapter of **Node.js at Scale** you are about to learn how the Node.js module system & CommonJS works and what does `require` do under the hood.

COMMONJS TO THE RESCUE

The JavaScript language didn't have a native way of organizing code before the ES2015 standard. Node.js filled this gap with the **CommonJS** module format. In this chapter we will learn about how the Node.js module system works, how you can organize your modules and what does the new ES standard means for the future of Node.js.

What is the module system?

Modules are the fundamental building blocks of the code structure.

The module system allows you to organize your code, hide information and only expose the public interface of a component using `module.exports`. Every time you use the `require` call, you are loading another module.

The simplest example can be the following using CommonJS:

```
// add.js

function add (a, b) {
  return a + b
}

module.exports = add
```

To use the `add` module we have just created, we have to require it.

```
// index.js
const add = require('./add')

console.log(add(4, 5))
//9
```

Under the hood, `add.js` is wrapped by Node.js this way:

```
(function (exports, require, module, __filename, __dirname) {
  function add (a, b) {
    return a + b
  }

  module.exports = add
})
```

This is why you can access the global-like variables like **require** and **module**. It also ensures that your variables are scoped to your module rather than the global object.

How does `require` work?

The module loading mechanism in Node.js is caching the modules on the first `require` call. It means that every time you use `require('awesome-module')` you will get the same instance of `awesome-module`, which ensures that the modules are singleton-like and have the same state across your application.

You can load native modules and path references from your file system or installed modules. If the identifier passed to the `require` function is not a native module or a file reference (beginning with `/`, `../`, `./` or similar), then Node.js will look for installed modules. It will walk your file system looking for the referenced module in the `node_modules` folder. It starts from the parent directory of your current module and then moves to the parent directory until it finds the right module or until the root of the file system is reached.

Require under the hood - `module.js`

The module dealing with module loading in the Node core is called `module.js`, and can be found in lib/module.js in the Node.js repository.

The most important functions to check here are the `_load` and `_compile` functions.

Module._load

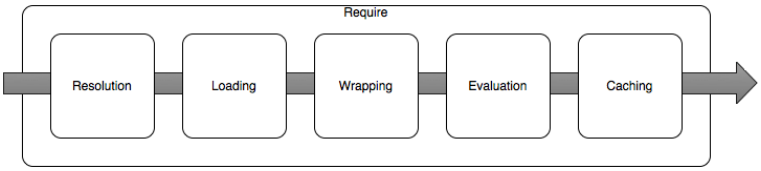
This function checks whether the module is in the cache already - if so, it returns the exports object.

If the module is native, it calls the `NativeModule.require()` with the filename and returns the result.

Otherwise, it creates a new module for the file and saves it to the cache. Then it loads the file contents before returning its exports object.

Module._compile

The compile function runs the file contents in the correct scope or sandbox, as well as exposes helper variables like `require`, `module` or `exports` to the file.



Pic: How Require Works - From James N. Snell

HOW TO ORGANIZE THE CODE?

In our applications, we need to find the right balance of cohesion and coupling when creating modules. The desirable scenario is to achieve **high cohesion and loose coupling** of the modules.

A module must be focused only on a single part of the functionality to have high cohesion. Loose coupling means that the modules should not have a global or shared state. They should only communicate by passing parameters, and they are easily replaceable without touching your broader codebase.

We usually export **named functions** or **constants** in the following way:

```
'use strict'

const CONNECTION_LIMIT = 0

function connect () { /* ... */ }

module.exports = {
  CONNECTION_LIMIT,
  connect
}
```

WHAT'S IN YOUR NODE_MODULES?

The `node_modules` folder is the place where Node.js looks for modules. **npm v2** and **npm v3** install your dependencies differently. You can find out what version of npm you are using by executing:

```
npm --version
```

npm v2

npm 2 installs all dependencies in a nested way, where your primary package dependencies are in their `node_modules` folder.

npm v3

npm3 attempts to flatten these secondary dependencies and install them in the root `node_modules` folder. This means that you can't tell by looking at your `node_modules` which packages are your explicit or implicit dependencies. It is also possible that the installation order changes your folder structure because npm 3 is non-deterministic in this manner.

You can make sure that your node_modules directory is always the same by installing packages only from a `package.json`.
In this case, it installs your dependencies in alphabetical order, which also means that you will get the same folder tree. This is important because the modules are cached using their path as the lookup key. Each package can have its own child `node_modules` folder, which might result in multiple instances of the same package and of the same module.

HOW TO HANDLE YOUR MODULES?

There are two main ways for wiring modules. One of them is using hard coded dependencies, explicitly loading one module into another using a `require` call. The other method is to use a dependency injection pattern, where we pass the components as a parameter or we have a global container *(known as IoC, or Inversion of Control container)*, which centralizes the management of the modules.

We can allow Node.js to manage the modules life cycle by using hard coded module loading. It organizes your packages in an intuitive way, which makes understanding and debugging easy.

Dependency Injection is rarely used in a Node.js environment, although it is a useful concept. The DI pattern can result in an improved decoupling of the modules. Instead of explicitly defining dependencies for a module, they are received from the outside. Therefore they can be easily replaced with modules having the same interfaces.

Let's see an example for DI modules using the factory pattern:

```
class Car {
  constructor (options) {
    this.engine = options.engine
  }

  start () {
    this.engine.start()
  }
}

function create (options) {
  return new Car(options)
}

module.exports = create
```

The ES2015 module system

As we saw above, the CommonJS module system uses a runtime evaluation of the modules, wrapping them into a function before the execution. The ES2015 modules don't need to be wrapped since the `import` / `export` bindings are created before evaluating the module. This incompatibility is the reason that currently there are no JavaScript runtime supporting the ES modules. There was a lot of discussion about the topic and a proposal is in `DRAFT` state, so hopefully we will have support for it in future Node versions.

To read an in-depth explanation of the biggest differences between CommonJS and the ESM, read the following article by James M Snell.

NEXT UP

We hope this chapter contained valuable information about the module system and how `require` works.

In the next volume of the Node.js at Scale series, we are going to take a deep dive and learn about the event loop, garbage collection and about writing native modules.

NODE.JS DEBUGGING AND MONITORING

At RisingStack, as an enterprise Node.js development and consulting company, we have been working tirelessly in the past two years to build durable and efficient microservices architectures with Node.js for our clients.

During this period, we had to face the cold fact that there aren't proper tools able to support the developers working with microservices architectures. **Monitoring, debugging and maintaining distributed systems is still extremely challenging.**

We want to change this situation!

This is why we created Trace - **a Node.js debugging and monitoring solution that treats microservices applications as first class citizens.**

If you're developing with Node.js, please check it out, it's free!