The npm Book

The Essential Guide to the Node Package Manager

Trevor Burnham

author of *Async JavaScript*

The npm Book

The Essential Guide to the Node Package Manager

Trevor Burnham

This book is for sale at http://leanpub.com/npm

This version was published on 2013-09-01

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Tweet This Book!

Please help Trevor Burnham by spreading the word about this book on Twitter!

The suggested hashtag for this book is #npmbook.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

https://twitter.com/search/#npmbook

Contents

Hello, npm!

In this chapter, we'll create our first Node project and add dependencies to it with npm. Even if you've used npm before, give it a skim; you might just pick up a few new tricks.

Installing Node and npm

If you haven't already, install the latest stable release of Node and npm so that you can follow along. Just go to http://nodejs.org/ and click "Download." For Windows and Mac folks, there's a handy GUI installer. If you're on *nix, I'll trust that you know how to install from source.

Managing multiple Node installations

If you're on a Mac or *nix system and you want to run different versions of Node for different projects, try nvm.[1]

After running the install script and starting a new shell session, use nvm to grab the latest stable version of Node and make it the default for new shell sessions:

```
nvm install 0.10
nvm alias default 0.10
```

Check your Node installation with `-v` (short for `--version`):

```
$ node -v
v0.10.9
```

As of this writing, the latest stable Node branch is 0.10.x, while 0.11.x is the unstable "bleeding edge" branch.

Unless you unchecked the option in the GUI installer (or used the `--without-npm` flag when installing from source), you should have npm installed as well. But if you don't, npm is famous for its one-line install:

[1] https://github.com/creationix/nvm

```
$ curl https://npmjs.org/install.sh | sh
```

Once installed, you can update npm with itself:

```
$ npm update -g npm
```

Use `-v` (`--version`) to check npm's version:

```
$ npm -v
1.2.25
```

Running `npm` by itself will give you a complete list of npm subcommands. You can run `npm help <subcommand>` to open that subcommand's `man` page.

Starting a project with `npm init`

Create a directory called `hello-npm` and `cd` into it. Now let's run our first npm command:

```
$ npm init
```

Helpfully, `npm init` explains what it does when you run it:

```
This utility will walk you through creating a package.json file.
It only covers the most common items, and tries to guess sane defaults.

See `npm help json` for definitive documentation on these fields
and exactly what they do.

Use `npm install <pkg> --save` afterwards to install a package and
save it as a dependency in the package.json file.

Press ^C at any time to quit.
name: (hello-npm)
```

So in short: `npm init` is a tool for creating (or modifying) a project's `package.json` file. We'll talk more about what `package.json` does in the next section, but first let's respond to `npm init`'s prompts.

The first thing we're being asked is what the name of our project should be. The default (shown in parentheses) is the name of the current directory, `hello-npm`. Let's accept the default by leaving the prompt blank and hitting Enter. We'll talk about good project names in the next section.

Here's how I answered all of the prompts:

```
name: (hello-npm)
version: (0.0.0)
description: A minimal Node project for The npm Book
entry point: (index.js)
test command:
git repository:
keywords: npm, example
author: Trevor Burnham
license: (BSD) MIT
```

I kept most of the defaults, but I added a description, a couple of keywords, and my name. I also opted for an MIT license instead of BSD (solely on aesthetic grounds; they're both fine open-source licenses). Feel free to improvise here. The only prompt that will matter for this chapter is the entry point, `index.js`.

After the prompts, `npm init` gives you a preview of the `package.json` it's about to create. For me, it looks like this:

```
{
  "name": "hello-npm",
  "version": "0.0.0",
  "description": "A minimal Node project for The npm Book",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [
    "npm",
    "example"
  ],
  "author": "Trevor Burnham",
  "license": "MIT"
}
```

Hit Enter to confirm, saving the `package.json`. You can always change it later, either by editing `package.json` directly or by running `npm init` again. `npm init` never removes information you've put in your `package.json`; it only changes what you want changed.

A look at `package.json`

A project's `package.json` fulfills several roles:

1. It provides npm with all kinds of information, as we'll see throughout this book.

2. It tells Node what to do with a package, as we'll see in the next section.
3. It gives humans a starting point when looking at a Node project.

Even at this stage, a Node developer looking at our `package.json` would learn a lot about our project. In addition to the obvious (name, description, keywords, and author), they can infer from the version number that it's a new project. Because of the default `scripts` that `npm init` gave us, they can also infer that we haven't implemented tests yet.

And most importantly, they'd infer that the entry point to our application is `index.js`. Of course, right now, there's no such file. We'll rectify that shortly.

There are many settings that npm recognizes in `package.json` beyond those that `npm init` offers to create. You can get a comprehensive list by running `npm help json`.

Creating the package entry point

Let's create a very simple script and save it as `index.js`:

```
// index.js
console.log('Hello, npm!');
```

We can run this script directly:

```
$ node index.js
Hello, npm!
```

We can also run it from another Node application using `require`. Let's enter the Node REPL, an interactive command-line environment:

```
$ node
> require('./index.js')
Hello, npm!
{}
```

(The `{}` is the return value of `require`. In this case, it's the `exports` object in `index.js`.)

Node allows us to omit the `.js` extension, so we can simply write `require('./index')`. If we run it again from the same REPL session, we'll get the same return value but not the `console.log` output, because `require` never runs the same script twice:

```
> require('./index')
{}
```

The `./` in `./index` is important, however. It tells Node that this path is relative to the current directory. If we don't explicitly give a relative or absolute path, Node will look for a matching module in a series of directories. More on that in the next chapter.

For now, let's back up a bit. In `package.json`, what does `"main": "index.js"` do? The answer is: When you `require` a directory containing a `package.json` pointing to a `main` script, Node acts like you `required` that script. So we can write:

```
> require('./')
{}
```

This also works for the `node` shell command:

```
$ node .
Hello, npm!
```

Pretty cool, right? Now, as it happens, we could do this without `package.json`, simply because our script is named `index.js`. That's the default script for a directory *unless* `package.json` says otherwise. Reiterating that default in our `package.json` just makes things clearer for us humans.

Our first dependency

Let's add some color to our string. We could look up ANSI codes to do that, but instead let's use a friendly library. A quick search of the npm registry with

```
$ npm search color
```

tells us that there are quite a few!

We want more than just names and descriptions, so let's hop over to npmjs.org/[2]. It turns out we don't even need to run a search, because right there in the "Most Depended On" list is a library called `colors`. Promising! Clicking that link takes us to npmjs.org/package/colors[3], where we find this succinct description:

> get colors in your node.js console like what

Sounds good! There's also a list of all the (many) packages in the npm registry that depend on `colors`, and a link to the library's GitHub page.

Let's install it as a dependency of `hello-npm`:

[2] https://npmjs.org/

[3] https://npmjs.org/package/colors

```
$ npm install --save colors
```

This will create a local directory called `node_modules` and download `colors` into it. And thanks to the `--save` flag, the dependency has been added to our `package.json` for us:

```
"dependencies": {
  "colors": "~0.6.0-1"
}
```

We'll learn all about dependencies in the next chapter. For now, let's rewrite `index.js` to take advantage of our new library. `colors` extends the `String` prototype so that we can express colored text very succinctly:

```
var colors = require('colors');
console.log('Hello,'.red, 'npm!'.green);
```

Run it, and you should get a Christmas-y twist on the original output.

The `node_modules` hierarchy

How did `require` know where to find the `colors` module? When you don't specify a relative or absolute path, Node first looks in the local `node_modules` directory, which is created when you `npm install` any package. When Node sees that `node_modules` has a subdirectory called `colors`, it takes that to be the `require` target. `node_modules/colors/package.json` has the tuple `"main": "colors"`, so Node runs `node_modules/colors/colors.js` and returns that script's `exports` object from our `require` function.

What would happen if there were no `node_modules/colors`? Node would look in `../node_modules/colors`, `../../node_modules/colors`, and so on up to the root of the filesystem. Then, finally, it would look in a handful of global folders.[4]

Finishing touches

I'd like to share `hello-npm` with the world, so I'm going to push it to GitHub and then publish the package to the npm registry. You won't be able to follow along with this section exactly, of course; there can only be one package named `hello-npm`. But if you create an npm registry account with

[4] I won't talk much about global modules, because they're widely considered a bad practice, except for distributing binaries. If you want the details, see the official docs: http://nodejs.org/api/modules.html#modules_loading_from_the_global_folders

```
npm adduser
```

and change the package name, you should have no problems.

First, I'll create a local git repo, commit everything, and push it to GitHub. Using git is beyond the scope of this book, so if you need a primer, check out Travis Swicegood's *Pragmatic Version Control Using Git*.[5] Of course, the git-fu involved here is minimal:

```
$ git init
$ git add -A
$ git commit
$ git remote add origin git@github.com:TrevorBurnham/hello-npm.git
$ git push -u origin master
```

Before I publish to npm, I should include a reference to the git repo in our `package.json`. Hey, wasn't that one of the prompts `npm init` gave us? Let me run it again…

```
name: (hello-npm)
version: (0.0.0)
git repository: (git://github.com/TrevorBurnham/hello-npm.git)
```

Nice! `npm init` automatically detected the remote origin I had set with git. Now I just accept the defaults, and my `package.json` has been extended with the GitHub repo's address:

```
"repository": {
  "type": "git",
  "url": "git://github.com/TrevorBurnham/hello-npm.git"
}
```

I'm ready to publish! We'll find out a lot more about publishing to the npm registry—authentication, versioning, unpublishing, interfacing programmatically—in the chapter entitled Publishing. For now, let's keep it simple:

```
$ npm publish
```

The output is verbose, but the long and short of it is, `hello-npm` version 0.0.0 now lives in the npm registry! We can now run

[5] http://pragprog.com/book/tsgit/pragmatic-version-control-using-git

```
$ npm install hello-npm
```

in any directory, on any npm-running machine in the world, and execute our script with a simple command:

```
$ node -e "require('hello-npm')"
```

Getting to know our package, remotely

We can use npm to learn more about packages without installing them. Let's take a look at the package we just created:

```
$ npm view hello-npm
```

will display all the info from our `package.json` and then some;

```
$ npm view hello-npm author versions
```

will display just the `author` and `versions` from the aforementioned infodump; and

```
$ npm docs hello-npm
```

will open the project's corresponding npmjs.org[6] page in our browser. (On operating systems other than OS X, you'll have to have the `google-chrome` utility installed.) If we wanted to make `npm docs` go somewhere else, we could set the `homepage` in `package.json`.

Here's one more:

```
$ npm issues hello-npm
```

That'll take us to the issue tracker at https://github.com/TrevorBurnham/hello-npm/issues[7]. Remember that the next time you want to complain to a package's author!

Conclusion

We've gotten our first taste of npm. We've seen how powerful even a simple `package.json` file can be, and how easy it is to install ordinary dependencies. In the next chapter, we'll look at how npm handles more complex dependency trees.

[6] https://npmjs.org

[7] https://github.com/TrevorBurnham/hello-npm/issues