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

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

Preface

In the last few years, Node.js has gone from a twinkle in Ryan Dahl's eye to a major phenomenon in the tech world. In 2009, Dahl made the first commit; in 2011, Node became the most-watched repo on GitHub, overtaking jQuery and Rails. That same year, the first NodeConf was "very, very sold out." Twitter and LinkedIn are using it. Microsoft is backing it. And although no 1.0 release is in sight as of this writing (Node 0.9 will be followed by Node 0.10), jobs for Node developers are opening up everywhere.

How did this happen? Node's meteoric rise is commonly attributed to a combination of two factors:

- **It's server-side JavaScript**. JavaScript has long been the *lingua franca* of the web. With Node, a developer can build a robust web application with just one language.
- **It's event-driven**. Node was designed from the bottom up for concurrent request handling. Tuning servers written in multi-threaded languages to minimize blocking takes work. With Node, non-blocking I/O is the default.

But this narrative overlooks a critical third factor: the surrounding ecosystem. From its inception, Node has been blessed with a community brimming with exceptionally smart developers with a passion for beautiful code. Wherever they saw a need (a database driver; a test framework; a command line argument parser), a robust open-source project would inevitably fill the gap.

Such was the case with package managers. Though Isaac Z. Schlueter's npm[1] wasn't the only attempt to provide easy distribution and dependency management for Node projects, it stood out above the rest. It grew to be such an ubiquitous utility that Node 0.4's `require` function was redesigned with npm in mind. And ever since Node 0.6.3, npm has been included by Node's installer.

In January 2012, Ryan Dahl stepped down as "gatekeeper" of the Node project and appointed Schlueter to replace him, stating:

> After three years of working on Node, this frees me up to work on research projects.

It was a fitting transition. Node was no longer just an interesting piece of software: It was the foundation for an entire community of developers increasingly concerned with getting their projects into production. And nothing epitomizes that community better than npm.

[1]http://npmjs.org/

Why this book?

There are already an overwhelming number of books on Node. But every one I've seen provides only a perfunctory introduction to npm.

In my opinion, knowing npm is as important to a Node developer as knowing your version control tool. Having the right subcommand at your fingertips will often make the difference between coding Zen and a searing headache.

Hence this book. *The npm Book* is intended to be a short but thorough guide, appropriate for Node beginners and experts alike.

Stylistic conventions

By now, you've probably noticed that *npm* is never capitalized. This is the official convention. As the npm FAQ[2] says (somewhat cheekily):

> Contrary to the belief of many, "npm" is not in fact an abbreviation for "Node Package Manager". It is a recursive bacronymic abbreviation for "npm is not an acronym".

Commands look like this:

```
$ npm -v
1.2.25
```

Unless otherwise specified, you can assume that all shell commands are being run from the base directory of a Node project.

A word for Windows users

npm fully supports Windows. However, I don't personally use Windows. So for simplicity's sake, I'll assume that you're on a *nix system. In practice, this mostly means that I'll be writing out paths in the form `/node_modules` while you'll see paths like `\node_modules`. However, if anything in this book is flat-out wrong under Windows, please let me know.

[2] http://npmjs.org/doc/faq.html#If-npm-is-an-acronym-why-is-it-never-capitalized

Feedback is welcome

If you find any errors or omissions in this book, or have any suggestions, I'd love to hear from you. You can reach me by email at trevorburnham@gmail.com, or on Twitter at @trevorburnham[3].

Now, let's get started!

[3]https://twitter.com/trevorburnham

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.[4]

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:

[4] 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/[5]. 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[6], 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`:

[5]https://npmjs.org/
[6]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.[7]

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

[7] 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.*[8] 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

[8]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[9] 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[10]. 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.

[9]https://npmjs.org
[10]https://github.com/TrevorBurnham/hello-npm/issues

Managing Dependencies

At first glance, supplying packages with the packages they need may seem like an easy problem. It's not. It's a problem that's long beguiled entire communities of smart, passionate programmers.

In the Ruby world (to take just one example), dependencies are installed with RubyGems (the `gem` command). Gems are installed at the system level: If you have several Ruby projects that all share the same gem, it only has to be installed once. Sounds great, right?

Usually, it is. But occasionally, it's a nightmare. When you have two projects that require different versions of the same gem, you have to jump through hoops to get both to work. What's worse is when one project depends on two things (call them dependencies X and Y), which in turn depend on different versions of dependency Z. *Ouch.*

Originally, npm took roughly the same approach as RubyGems. That changed in npm 1.0, released in March 2011. Since then, npm's mantra has been: "Local, local, local." Unless you tell it otherwise, npm only installs to the current directory's `node_modules` subdirectory. And there's more: If you install a dependency with dependencies of its own, those subdependencies are installed in their parent's `node_modules` directory—and so on, all the way down.

In this chapter, we'll learn the ins and outs of `npm install`. We'll also learn about keeping track of a project's dependencies with `package.json`, symlinking projects for local development, and sharing a dependency graph with `npm shrinkwrap`.

Listing dependencies

In the last chapter, we added the `colors` dependency to our project with

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

(where `--save` added the dependency to our `package.json` for us). Running `npm ls` (`ls` is short for `list`) from that project's directory gives us:

```
$ npm ls
hello-npm@0.0.0 /Users/trevor/hello-npm
└── colors@0.6.0-1
```

The output represents a very simple tree: `hello-npm` (version 0.0.0) is the parent of `colors` (version 0.6.0-1).

Let's look at a slightly more complex example. Create a new directory (let's call it `express-project`) and run:

```
$ npm install express
```

Now `npm ls` has all kinds of information for us:

```
$ npm ls
/Users/trevor/express-project
└─┬ express@3.0.0beta6
  ├── commander@0.6.1
  ├─┬ connect@2.3.8
  │ ├── bytes@0.1.0
  │ ├── cookie@0.0.4
  │ ├── crc@0.2.0
  │ ├── formidable@1.0.11
  │ └── qs@0.4.2
  ├── cookie@0.0.3
  ├── debug@0.7.0
  ├── fresh@0.1.0
  ├── methods@0.0.1
  ├── mkdirp@0.3.3
  ├── range-parser@0.0.4
  ├─┬ response-send@0.0.1
  │ └── crc@0.2.0
  └─┬ send@0.0.2
    └── mime@1.2.6
```

The tree tells us that we depend on `express`, which in turn has `10` direct dependencies, some of which have sub-dependencies.

Extended listings

What do all of these packages do, you ask? You can use `ll` (or its alias `la`) instead of `ls` to get more verbose output. Specifically, `npm ll` reads each dependency's `package.json` and gives you its description and repo location, when provided:

```
$ npm ll
│ /Users/trevor/express-project
│
└─┬ express@3.0.0beta6
  │ Sinatra inspired web development framework
  │ git://github.com/visionmedia/express
  ├── commander@0.6.1
  │   the complete solution for node.js command-line programs
  │   https://github.com/visionmedia/commander.js.git
  ├─┬ connect@2.3.8
  │ │ High performance middleware framework
  │ │ git://github.com/senchalabs/connect.git
  │ ├── bytes@0.1.0
  │ │   byte string parser (5mb etc)
...
```

Machine-friendly listings

`npm ls` accepts a `--parseable` flag (`-p` for short), which tells it to display its output as a list of directory names instead of a tree:

```
$ npm ls -p
/Users/trevor/express-project
/Users/trevor/express-project/node_modules/express
/Users/trevor/express-project/node_modules/express/node_modules/commander
/Users/trevor/express-project/node_modules/express/node_modules/connect
/Users/trevor/express-project/node_modules/express/node_modules/connect/node_modu\
les/bytes
...
```

You can also use the `--parseable` flag with `npm ll` to make it add the package name and version number after each directory name:

```
$ npm ll -p
/Users/trevor/express-project::
/Users/trevor/express-project/node_modules/express:express@3.0.0beta6:
/Users/trevor/express-project/node_modules/express/node_modules/commander:command\
er@0.6.1:
/Users/trevor/express-project/node_modules/express/node_modules/connect:connect@2\
.3.8:
/Users/trevor/express-project/node_modules/express/node_modules/connect/node_modu\
les/bytes:bytes@0.1.0:
...
```

Missing and extraneous dependencies

Normally, a project's `package.json` should be in sync with its `node_modules` directory, and vice versa. When a dependency is listed in `package.json` but not installed in `node_modules`, it's said to be *missing* or *unmet*. Conversely, when a package is installed in `node_modules` but not listed as a dependency in `package.json`, it's said to be *extraneous*. We can use `npm ls` to identify both of these inconsistencies.

Right now, our `express-project` directory has no `package.json`. Let's create one by running

```
$ npm init
```

and accepting the defaults by hitting Enter when prompted. The result should look like this:

```
{
  "name": "express-project",
  "version": "0.0.0",
  "main": "index.js",
  "dependencies": {
    "express": "~3.0.0beta6"
  },
  "devDependencies": {},
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "license": "BSD"
}
```

Notice that `npm init` automatically scanned `node_modules` and brought in our `express` dependency. If we run `npm ls`, we'll get the same output as before, except with our project name and version number at the top.

Let's add a fake dependency to `package.json` to see what happens:

```
...
"dependencies": {
  "express": "~3.0.0beta6",
  "fake-dependency": "1.2.3"
},
...
```

(Be sure to add that comma at the end of the `express` line. Otherwise, the file will be invalid JSON and npm will refuse to parse it.)

Now run

```
$ npm ls
```

again. The output will be the same as before, except with one extra line:

```
└── UNMET DEPENDENCY fake-dependency 1.2.3
```

In a real project, we would fix this inconsistency by either removing the dependency from `package.json` with `npm uninstall --save fake-dependency` or installing it with `npm install fake-dependency`. Or we could run `npm update` to bring in *all* unmet dependencies.

Now let's try removing the `dependencies` section of our `package.json` entirely. All npm needs for a `package.json` to be valid is a name and version, like this:

```
{
  "name": "express-project",
  "version": "0.0.0"
}
```

Run

```
$ npm ls
```

one more time. The output will look just like it did when `package.json` included the `express` dependency, except for the word "extraneous":

```
express-project@0.0.0 /Users/trevor/express-project
└┬ express@3.0.0beta6  extraneous
...
```

In a real project, we would fix this inconsistency by either adding `express` to `package.json` with `npm install --save express` or removing it from `node_modules` with `npm uninstall express`.

Specifying dependency versions

npm assumes that package versions obey the rules of SemVer[11] ("semantic versioning"), with a few small modifications. (If you're interested in the details, check out the documentation for Isaac Schlueter's node-semver[12], the module that powers npm's version number parsing.)

Semantic versioning is pretty straightforward: Everything has three version numbers of the form `major.minor.patch`. semver.org recommends that patch numbers only be used for bug fixes, minor numbers for backward-compatible changes, and major numbers for backward-incompatible changes. Of course, this convention isn't enforced, but Node does require that all version numbers contain at least those three components.

If there's an additional component, optionally separated from the patch number with a `-`, then the version number is considered to be a *pre-release*. So `3.0.0beta6 < 3.0.0`.

In the last section, we saw that `npm init` automatically detected a module we'd installed (`express`) and added it to our `dependencies` in `package.json`:

```
...
"dependencies": {
  "express": "~3.0.0beta6"
},
...
```

Why the ~? That tells npm that we want the latest version with the same major and minor version numbers. We could accomplish the same thing with a range:

```
...
"dependencies": {
  "express": ">=3.0.0beta6 <3.1.0"
},
...
```

Of course, knowing that `3.0.0beta6` exists, we could omit the `>=3.0.0beta6` part of the range. We could also simply use a wildcard `x` in place of the patch number:

[11]http://semver.org/

[12]https://github.com/isaacs/node-semver

```
...
"dependencies": {
  "express": "3.0.x"
},
...
```

All these ways of specifying a version work equally well on the command line, where a version or a range can be used after an `@` in `npm install`:

```
$ npm install express@">=3.0.0beta6 <3.1.0"
```

Tagged versions

Package authors may choose to tag any version of their package with any unique string. (More on that in the "Publishing Packages" chapter.) A tag is simply an alias for a version number, just as a tag in Git is an alias for a commit SHA.

The syntax for installing a tagged version of a package is what you might expect: Swap in the tag for the version number. For example, to install the version of express tagged "latest":

```
$ npm install express@latest
```

As it happens, this is how npm interprets an install command with no version specifier:

```
$ npm install express
```

The default tag is actually a configurable setting, though one you're unlikely to want to change:

```
$ npm config get tag
latest
```

The same default is used if a project's `package.json` specifies a dependency with no version specifier:

```
...
"dependencies": {
  "express": ""
},
...
```

On the publishing side, npm enforces tagging the last published version of a package as "latest" by default. However, package authors can choose to tag an older version as "latest," typically when a new version is experimental.

Installing from a Git repo

npm supports installing packages directly from any Git repository that you have access to. This is handy when you want to install a package that isn't published in the npm registry, or a forked variant of one that is.

For example, to install express directly from the project's master branch on GitHub, you would write:

```
npm install git+https://github.com/visionmedia/express
```

(The `+https` part is optional, but recommended if the host server supports it.)

If you don't want to install from master, you can add a specific branch, tag, or commit SHA to install:

```
npm install git+https://github.com/visionmedia/express#3.0.0rc4
```

You can use Git references in a `package.json` as well:

```
...
"dependencies": {
  "express": "git+https://github.com/visionmedia/express#3.0.0rc4"
},
...
```

If your company uses hosted Git repositories, you might want to consider using this feature to distribute packages internally. Alternatively, you can set up your own registry. (More on that in the Using npm internally chapter.)

GitHub repo shorthand

Given the massive popularity of GitHub relative to other Git hosting services, npm provides an abbreviated syntax that parallels GitHub's url scheme: `npm install author/repository`. So for example,

```
npm install jashkenas/coffee-script
```

is shorthand for

```
npm install git+https://github.com/jashkenas/coffee-script
```

As with the full Git syntax, a branch, tag, or SHA to install from instead of master may be added after a `#`:

```
npm install jashkenas/coffee-script#issue2120
```

Linking a local package for development

Suppose you want to work on a project and one of its dependencies at the same time. You'd like to be able to make changes to the dependency and see their effect on the base project right away.

One approach would be to install the dependency normally, then make changes to the installed copy in the base project's `node_modules`. But that may be difficult, because what npm installs is only a subset of the original project. (For example, if there's documentation and a test suite, it probably isn't part of the npm package.)

Another approach would be to clone the dependency's git repo directly into the base project's `node_modules`. That's a little better, but you could lose that directory and all the work you've done with a single accidental `npm uninstall`. If only the operating system gave you a way to keep that directory in one place while pretending that it also lives elsewhere…

Aha! Symlinking! You could symlink the directory yourself, but npm makes it far easier:

1. In the dependency's directory, run `npm link`. This creates a symlink from npm's global module directory to that directory.
2. In the base project's directory, run `npm link <dependency>`. This creates another symlink from `node_modules/<dependency>` to the global module path.

That's it. As a bonus, `npm link` will also install all upstream dependencies and copy any executable scripts in the dependency to the global executable directory, removing the need for `npm install -g`.

You can use `npm ls` in the base project to confirm that the symlinked dependency is being used and, if you've forgotten, see where the symlink points to:

```
$ npm ls
base-project@0.0.0 /Users/trevor/base-project
└── dependency@1.2.3 -> /Users/trevor/dependency
```

To remove the symlinked dependency, either `npm unlink dependency` or `npm uninstall dependency` will do the trick. If you run `npm install dependency`, you'll replace the symlink with a standard installation.

Other dependency types

As far as `npm install` is concerned, every package listed in a project's `dependencies` is absolutely essential for that package to be useful. But npm also supports two special dependency types that are only installed under specific circumstances.

Development dependencies

The quintessential example of a non-essential dependency is a testing library. If you're developing Express, you need to be able to run its tests. But if you're developing one of the thousands of projects that depends on Express, you probably don't. That's why `package.json` also allows you to specify `devDependencies`. The syntax is the same as `dependencies`, but these are only installed when:

1. You run `npm install` or `npm link` from the base of the project, or
2. You run `npm install --dev`.

In other words, `devDependencies` in your project's dependencies are ignored unless you use the `--dev` flag.

npm provides a special syntax for installing a development dependency and adding it to your `package.json`. Whereas

```
$ npm install <package> --save
```

would add that package to `dependencies`,

```
$ npm install <package> --save-dev
```

will add it to `devDependencies`.

Optional dependencies

`npm install`/`npm update` will error out if any package specified in `dependencies` can't be found. Every now and then, you might have a dependency that you only want npm to try to install. The most common example is a dependency that uses native, OS-specific code to provide features or enhanced performance. These are specified in `package.json` in a hash named `optionalDependencies`.

Use the `--save-optional` flag when installing a dependency to tell npm to add it to the package's `optionalDependencies`:

```
$ npm install <package> --save-optional
```

Bundling dependencies

You can choose to "bundle" dependencies, distributing their files along with your package when you run `npm publish`. This is a very handy tool for deployment: If you publish a package with all of its dependencies bundled, then the server only needs to download and unzip one package, rather than a whole tree of packages. Not only is that efficient, it also prevents troublesome version inconsistencies. Node hosting services like Nodejitsu use this feature heavily.

To bundle a dependency, you need to list it in `dependencies` as usual and add its name to the `bundledDependencies` array. For example, if we wanted to bundle the sole dependency of our `hello-npm` project from the previous chapter, our `package.json` would look like this:

```
...
"dependencies": {
  "colors": "~0.6.0-1"
},
"bundledDependencies": [
  "colors"
],
...
```

Shrinkwrapping dependencies

We've seen how `package.json` allows you to loosely specify package versions, with specifiers like "~2.0.0" or "latest." But sometimes, you want to ensure that everyone who runs `npm install` has the exact same set of dependencies. Merely using exact version numbers in your `package.json` isn't enough, because some of those packages or packages further up the tree may use looser version specifiers.

Luckily, npm allows you to take a snapshot of which dependencies a project actually has installed with the `shrinkwrap` command:

```
$ npm shrinkwrap
wrote npm-shrinkwrap.json
```

The `npm-shrinkwrap.json` file the command creates fully specifies all dependencies, including upstream dependencies. When `npm install` sees a project with both a `package.json` and an `npm-shrinkwrap.json`, the `npm-shrinkwrap.json` takes precedence.

Provided that no package authors publish two variations of the same package with the same version number, or unpublish a package that you depended on, running `npm install` in a shrinkwrapped project's directory will give you an exact copy of its `node_modules` directory from when it was shrinkwrapped. (There is one exception: If any modules are listed in `devDependencies`, `npm shrinkwrap` will exclude them unless you pass the `--dev` flag.)

For some, `npm shrinkwrap` is a tool for deployment. This is a little risky, though, as it leaves your ability to deploy new instances subject to the whims of the npm registry. Bundling all of your production dependencies (see the previous section) is a much safer approach. In practice, shrinkwrapping is more of a debugging tool. When a developer encounters a bug that seems to be related to a dependency, swapping shrinkwrap files with their colleagues can make it much easier to pin down the source.

Conclusion

That does it for our coverage of installing dependencies with npm. Our next chapter will take us to the other side: publishing a package to the npm registry.

Publishing Packages

Sometimes, your code is too good to just keep it to yourself. When the urge to share strikes, your new best friend is

```
$ npm publish
```

Of course, there's more to publishing than that. You'll need to create an npm account, to secure your published work. You might also want to share publishing privileges with collaborators. Once your package's authorship is secure, you'll want to maintain it responsibly with accurate version tags and deprecation warnings for buggy old versions. Do all of that, and the JavaScripters of the world will thank you.

Managing your npm registry account

npm takes a very utilitarian approach to account management. In fact, npm has only two distinct commands for dealing with with authentication, registration, and account administration: `adduser` and `whoami`.

Joining npm userland

If you don't have an account in the npm registry already, start by running

```
$ npm adduser
```

You'll be prompted to choose a username, a password, and an email address. Your username is permanent, so choose wisely or you'll have to create another account later.

Be aware that the email address you enter here will be public, displayed on your profile page at https://npmjs.org/∼yourusername. If you're secretive with your primary email address, you'll want to use a secondary one, but be sure to associate it with a good Gravatar[13]—that's displayed on your profile page, too.

Logging in and out

To see which account npm is using on your machine, run

[13]http://gravatar.com/

```
$ npm whoami
Not authed.  Run 'npm adduser'
```

Like the message says, if you aren't already logged in, `npm adduser` is the command to run. (In fact, it's aliased as `npm login`.) You still need to enter all of the same information as when you created the account. Whatever you enter as your email address when logging in will overwrite the existing email address on file.

npm doesn't have a logout command, but you can clear your information directly. Run `npm config list` and notice the `userconfig` section:

```
; userconfig /Users/trevor/.npmrc
email = "trevorburnham@gmail.com"
username = "trevorburnham"
```

That tells you which file contains your user information. You won't actually see your username in there; instead, your username and password are encrypted together in a single value called `_auth`. Delete it, and you'll be back to the "Not authed" state.

Changing your account information

Once you've created an npm account, you can log in to your "home" on the npm website at https://npmjs.org/~[14]. There, you can change your email or password. You can also add more information to your profile page. Don't worry: npmjs.org has no aspirations to be the next LinkedIn.

Preparing your package for publication

Before you start streaming bits up to the npm registry like a *Matrix* screensaver in reverse, you'll want to give your package a little extra love and care.

Making your `package.json` complete

At a minimum, a project's `package.json` needs to have a `name` and a `version`. For example:

```
{
  "name": "hello-npm",
  "version": "0.0.0"
}
```

[14]https://npmjs.org/~

The only requirement for the `name` field is that it's a URL-safe string unique to the registry you're publishing to. As a stylistic matter, it should be all-lowercase, with words separated by hyphens (not underscores). Also, the use of `"js"` or `"node"` in a name is redundant. For the same reason, don't use version numbers in the name.

Name squatting

Because package names are a finite resource, conflicts can arise. If you sit on a desirable name (by publishing a package with that name which no one is interested in using), you may be asked to forfeit it. Publishing a package just to reserve a name is heavily discouraged.

Here's the good news: If someone else has already published something with the name you want, but their package is little-used, you have some recourse. Run `npm help disputes` for details.

The `version` must be SemVer-parseable. That means three numbers (major, minor, and patch)separated by `.`s. If you're interested in learning all the rules, you can read the SemVer 2.0.0 spec[15]. As a matter of convention, your package should have a major version number of `0` until you consider it ready for production use.

It's common for the version number in your `package.json` to correspond to a git tag. You can keep both in sync using the `npm version` command. More on that in the Versioning your package section, later in this chapter.

Here are some other fields you should strongly consider adding to your `package.json` before publication, for the benefit of people trying to find or understand your package:

`description`

A sentence or two, shown prominently on your package's page (https://npmjs.org/package/yourpackage).

`keywords`

An array of words people can use with the `npm search` command, or the searchbox at npmjs.org, to find your package. Note that npm searches all words in the description field as well. Having a good description is more important than having keywords.

`repository`

The location of your project's version control repository. For example:

[15]http://semver.org/spec/v2.0.0.html

```
"repository" : "git://github.com/TrevorBurnham/hello-npm.git"
```

If your project lives on GitHub, npm is smart enough to infer its homepage and bug tracker from this URL as well. If you don't use GitHub for those things, you should set the `homepage` and `bugs` fields separately.

`author` **or** `contributors`

npm has two way of specifying people associated with a package: A single `author`, and an array of `contributors`. Each person is specified as a string of the form `"name <email> (url)"`. For example, here's how npm's own `package.json` lists its author:

```
"author": "Isaac Z. Schlueter <i@izs.me> (http://blog.izs.me)"
```

`contributors` uses the same format, but with an array of people. The two fields are not exclusive. A `package.json` can list a single author plus an array of contributors, or it can have an array of contributors but no author.

Adding a license

There are two ways to specify a package's copyright license with npm. The first and easiest is to add a `license` field to the `package.json`, with either a URL pointing to a license or the name of a license. The name should be one of the licenses listed at http://opensource.org/licenses/alphabetical[16]. For example, npm's own `package.json` specifies:

```
"license": "Artistic-2.0"
```

On the package's npmjs.org page[17], this generates a link to http://opensource.org/licenses/Artistic-2.0[18].

The second way to specify a license is with a traditional `LICENSE` file, containing the full legal text of the license. Many packages include this in addition to the `package.json` field, so that the license is both visible at a glance and distributed with the code. Of course, if you do both, you'll want to double-check that the same license is specified in both places.

[16]http://opensource.org/licenses/alphabetical
[17]https://npmjs.org/package/npm
[18]http://opensource.org/licenses/Artistic-2.0

Adding a README

npm will complain, loudly, if your package doesn't have a README file with at least a line of text. It can have an extension (`README.txt`, `README.md`), but it must be included among your published files, or else everyone who installs your package will be subjected to a warning about its lack of minimal documentation.

If your project is on GitHub, its README doubles as its homepage, so you have twice the incentive to write one.

Choosing which files to publish

When you run `npm publish`, npm will create a tarball of your project and upload it to the npm registry. But, it can exclude certain files. You can specify those files by creating a `.npmignore` file. The format of a `.npmignore` is the same as a `.gitignore`: Each line contains a file/directory name, with `*` treated as a globbing wildcard. For example,

```
# .npmignore
tmp/
*.log
```

will ignore the `tmp` directory and all files with the `.log` extension. For more information on `.gitignore` syntax, I recommend the "Ignoring Files" section of *Pro Git*[19].

If no `.npmignore` file exists but a `.gitignore` does, the `.gitignore` will be used by npm as if it were named `.npmignore`.

There are a handful of files and directories that npm ignores by default, including `.git` and, importantly, `node_modules`. For a complete list, see `npm help developers` under the heading "Keeping files."

Versioning your package

As we saw in Making your `package.json` complete, your package needs to have a version of the form `x.y.z` before it can be published. This, along with your package name, gives each tarball you upload with `npm publish` a unique identifier. If two people run

```
npm install your-package@1.2.3
```

they're guaranteed to install the exact same thing—unless you've used the dreaded `npm publish --force` to overwrite the existing `1.2.3` version of `your-package` in the registry.

But version numbers are more than just identifiers. They can advertise a package's stability and maturity, or the reverse.

[19]http://git-scm.com/book/en/Git-Basics-Recording-Changes-to-the-Repository#Ignoring-Files

npm version

Manually editing the version number in `package.json` can grow tedious. Fortunately, npm provides the `npm version` command. In its simplest form, it replaces the version number in `package.json` with a new one:

```
$ npm version 1.0.0
v1.0.0
```

Better yet, the command can do semantic version incrementing for you. If you've made *major* changes that could break project's relying on the previous version of your package, run `npm version major`:

```
$ npm version major
v2.0.0
```

If you've made *minor* changes that add new functionality, run `npm version minor`:

```
$ npm version major
v2.1.0
```

If you've written a *patch* to fix a bug, run `npm version patch`:

```
$ npm version patch
v2.1.1
```

Pre-release versions

The SemVer spec allows pre-release versions to be specified by adding a hyphen and a string. For example, `3.0.0-beta1` is considered semantically earlier than `3.0.0`.

The command `npm version build` lets you increment these numbers:

```
$ npm version build
v3.0.0-beta2
```

`npm version` does one more thing for you: If your project is in a Git repository, it will create a commit for you with the `package.json` change, and tag that commit with the new version. By default, the commit message is just the version number. You can specify a commit message with the `-m` flag, using `%s` as a placeholder for the version number:

```
$ npm version patch -m "Bump to %s for flux capacitor fix"
v2.1.2
$ git log -1 v2.1.2
commit dae4183c1ac9c738f3d2c2a8b4bab02503e0c63c
Author: You <you@example.com>
Date:   Fri Dec 21 19:51:26 2012 -0400

    Bump to 2.1.2 for flux capacitor fix
```

Maintaining a solid git history will make it easier to track down bugs (particularly with `git bisect`). It's a good practice to keep a tag in your Git repository corresponding to each version you've published to the npm registry.

Pushing tags to GitHub

By default, `git push` does not push tags. With Git 1.8.3+, you can use `git push --follow-tags` to push commits and their related tags in a single command. Otherwise, you'll have to use the separate command `git push --tags`, which pushes *only* tags.

Adding more information to the registry

At this point, you know everything you need to know to publish your package to the npm registry. But there are a few post-publishing operations, aside from the self-explanatory (and heavily discouraged) `npm unpublish`, that you should be aware of.

Tagging versions

In Tagged versions, we saw that npm has its own notion of tags, separate from (though analogous to) Git's. npm tags live only in the registry, and a version must be published before it can be tagged. A tag simply works as an alias for a version number:

```
$ npm tag hello-npm@0.0.0 first
$ npm install hello-npm@first
hello-npm@0.0.0 node_modules/hello-npm
```

By default, every time you publish a new version of your package, the new version is tagged with the value of the `tag` config, which defaults to `"latest"`. And conversely, when someone installs a package without specifying a version, the version specified by their `tag` config is used.

It's a good practice to avoid tagging unstable, pre-release versions as `"latest"`. You can do this by temporarily changing your config, but a better way is to add this to your `package.json`:

```
publishConfig: {
  tag: "beta"
}
```

Settings in `publishConfig` override the corresponding npm configs when you run `npm publish`.

If you aren't sure how your package is tagged in the registry, you can check by running `npm info <package> dist-tags`.

Deprecating versions

After fixing a major bug, you can do more than just release a patch. You can actively discourage people from installing older versions of your package. To do that, run:

```
$ npm deprecate your-package@"<1.2.3" "v1.2.3 fixed memory leak"
```

(Any version range syntax will work. See Specifying dependency versions for more information.)

Now anyone who tries to install a deprecated version of your package from the registry will see your warning.

Conclusion

You now have everything you need to create and distribute the world's next big hit package! You've learned how to talk to the npm registry, what it needs from you, and how to keep your package neat and tidy for your users.

In the next chapter, we'll learn how to keep packages in our own internal registry, where our secret code can be safely locked away.

Using npm Internally

While npm is geared toward sharing code publicly, with a few tweaks, it can be adapted for use within a team. In this chapter, we'll see how you can set up your very own private npm registry, allowing you to keep certain packages between you and your colleagues.

Before we start, a disclaimer: Although running your own repository has many benefits, it's not for everyone. If you just want to keep a handful of packages within your company, and you already use Git repositories for those projects, it might be easier to tell npm to download those packages directly from those repos:

```
"dependencies": {
  "my-package": "git+ssh://git@example.com:my-package.git#v1.0.0"
}
```

Preventing package publication

The easiest way to prevent a package from finding its way to the public npm registry is to mark it as private. In its `package.json`:

```
"private": true
```

This will cause the `npm publish` command to simply fail:

```
$ npm publish
Error: This package has been marked as private
```

If you only want a package to be published to your internal registry, once you've set that up, add this to your `package.json` instead:

```
"publishConfig":{
  "registry": "http://registry.example.com"
}
```

Setting up a registry

In principle, any service that knows how to respond to queries from the npm client can act as an npm registry. There's even an npm package, reggie[20], that acts as a self-contained registry server. But the standard way to create a registry is to use CouchDB. You could set up a CouchDB database with the appropriate schema from scratch by following the instructions at https://github.com/isaacs/npmjs.org[21]. But in this section, we'll take a simpler approach: replicating the public registry. As we'll see, replication not only saves us some trouble, it also lets us use a single registry for distributing all packages, public and internal alike.

Creating the database

From a technical standpoint, the npm registry is just a CouchDB instance. CouchDB has an HTTP API, which makes it very easy to talk to. In fact, if you visit http://registry.npmjs.org/[22], you're making a query to the same CouchDB instance that npm itself does.

The CouchDB install process varies from one system to another, so head to http://couchdb.apache.org/[23] for details. npm requires at least version 1.1.0, and the latest stable version is recommended.

Hosted CouchDB

If you don't feel like doing your own system administration, you can get a hosted CouchDB instance from IrisCouch[a], home of the public npm registry.

[a]http://www.iriscouch.com/

Once CouchDB is running on your target machine, you should be able to send it commands using over HTTP. I'll assume you have curl[24] installed. Let's test that CouchDB is alive:

```
$ curl http://localhost:5984/
{"couchdb":"Welcome","uuid":"eed13e127ba8bc65da47e647d39ab4b6","version":"1.3.1"}
```

(If CouchDB is running on another machine/port, change `http://localhost:5984` to that address throughout this chapter.)

[20]https://github.com/mbrevoort/node-reggie
[21]https://github.com/isaacs/npmjs.org
[22]http://registry.npmjs.org/
[23]http://couchdb.apache.org/
[24]http://curl.haxx.se/docs/manpage.html

Replicating the public registry

npm was designed with a one-registry-to-rule-them-all mindset. It cannot, for example, query multiple registries when you run a command like `npm install`. Fortunately, you don't have to choose between the wealth of open source packages on offer at the public registry and the internal packages in your own registry. Replication gives us the best of both worlds, at the expense of some storage space and bandwidth (cheap commodities these days). As a bonus, we'll have access to the latest public package versions even when `registry.npmjs.org` goes down.

Without further ado, here's the command:

```
$ curl -X POST -H "Content-Type: application/json" http://localhost:5984/_replica\
tor -d '{"user_ctx": {"name":"replicator", "roles":["_admin"]}, "source":"http://\
isaacs.iriscouch.com/registry/", "target":"registry", "create_target":true, "cont\
inuous":true}'
{"ok":true,"id":"5504ff34212d93d09185b10c01000e31","rev":"1-9fee84cd1d78a991d988f\
6b314584023"}
```

Right away, your CouchDB will start reading the contents of the public npm database. Every single package will (eventually) be downloaded to your server. The `"continuous": true` flag tells CouchDB that we don't just want to just copy a snapshot of the database; we want to download new data every time there's an update, *forever.*

Note that although `registry.npmjs.org` is typically synonymous with `isaacs.iriscouch.com/registry`, it's important to use the latter address with this particular command.

Replication in older CouchDB versions

Prior to CouchDB 1.3, the preferred command was `_replicate` rather than `_replicator`, and the `user_ctx` parameter was unnecessary:

$ curl -X POST -H "Content-Type: application/json" http://localhost:5984/_replicate -d '{"source":"http://isaacs.iriscouch.com/registry/", "target":"registry", "create_target":true, "continuous":true}'

Installing the registry app

CouchDB has the ability to run JavaScript code, which is how it runs the logic needed to be a fully functional registry. Unfortunately, that code isn't copied during replication. So, we need to install it from the https://github.com/isaacs/npmjs.org[25] project:

[25] https://github.com/isaacs/npmjs.org

```
$ git clone https://github.com/isaacs/npmjs.org.git
$ cd npmjs.org
```

The next step is to push some of the code from this repo to our CouchDB app. There's an npm library, `couchapp`, that makes this very simple. Since we're using it as a binary, we'll want to install it as a global:

```
$ npm install -g couchapp
$ couchapp push registry/app.js http://localhost:5984/registry
```

Now your new registry is officially alive! You should be able to use it as the registry in your npm client, albeit with an unnecessary verbose URL (a problem we'll rectify shortly):

```
$ npm set registry http://localhost:5984/registry/_design/scratch/_rewrite
```

One more thing: The web interface behind https://npmjs.org/[26] lives in that same repository. If you want it, you can install it the same way you installed the core app:

```
$ couchapp push www/app.js http://localhost:5984/registry
```

However, note that you'll have to figure out for yourself how to secure that website to prevent outsiders from viewing information about your internal packages.

Using a domain name

In order to get a nice clean URL, we'll have to make a few changes to the CouchDB configuration file (typically in `/usr/local/etc/couchdb/local.ini`). First, we'll have to change a security setting in the `[httpd]` section:

```
[httpd]
secure_rewrites = false
```

If `true` (the default), this setting would prevent the rewrite rules we're about to add from working.

Second, we'll have to add a vhost:

[26]https://npmjs.org/

```
[vhosts]
localhost:5984 = /registry/_design/scratch/_rewrite
```

If you have a domain like `registry.yourdomain.com` pointing at your machine, use that instead of `localhost`. You can also add a domain for the web interface, if you chose to add it:

```
[vhosts]
registry.yourdomain.com:5984 = /registry/_design/scratch/_rewrite
npm.yourdomain.com:5984 = /registry/_design/ui/_rewrite
```

Restart CouchDB for the configuration changes to take effect. Now, at last, you can set a sane-looking registry URL in npm:

```
$ npm set registry http://localhost:5984/registry/
```

Want to get rid of the port number? If you have the requisite user privileges, you can modify `local.ini` to make CouchDB run on the standard HTTP port:

```
[httpd]
port = 80
```

Securing your registry

First, let's stop allowing anyone with our registry's address to send CouchDB administrative commands:

```
[admins]
admin = <password>
```

Second, if you don't want to rely on knowledge of your registry's address alone to keep your packages secure, you'll need to deny anyone who isn't a recognized user from performing standard npm requests against your registry. Unfortunately, this is very tricky to do, and the process is in flux (see npm issue 106[27] among others). Some private registries work around this by having a single CouchDB user with all requisite privileges for reading and writing the `registry` schema, and distributing that user's credentials throughout the team (to be entered in `npm adduser`). This works, but it's obviously not a fine-grained security solution. Unfortunately, there isn't much of an alternative as of this writing. Note that you will need to set

[27]https://github.com/isaacs/npmjs.org/issues/106

```
$ npm set always-auth true
```

for this to work, so that npm sends the CouchDB user's credentials even for read-only commands like `npm install`.

If you have an SSL cert for your domain, you can set that up in the `[ssl]` section of your `local.ini` and change the port to 443, allowing your registry to live at `https://registry.yourdomain.com` and preventing your repository from being compromised by man-in-the-middle attacks. You will, however, have to add the cert to each npm client that uses the repository with the `npm set ca` command.

Conclusions

npm is a remarkably transparent project, and that shows with how easy it makes it to not only set up private registries but to copy the main registry in its entirety. Even if the main registry's datacenter were to be struck by a meteor tomorrow, the many private registries receiving continuous replication ensure that npm ecosystem would survive.

Scripting with npm

Acronym or not, npm is a package manager first and foremost. Up to this point in the book, everything we've done with npm has revolved around packages. But there's another side to npm: It's a script runner.

Because every node project has a `package.json`, it's a convenient place to store small, simple shell scripts (possibly referencing big, complex scripts that live elsewhere in the project). The most common example of this is a script that runs the project's test suite. You might recall that `npm init` even gave us a placeholder for such a script in our project's `package.json`:

```
...
"scripts": {
  "test": "echo \"Error: no test specified\" && exit 1"
},
...
```

We can run this script with the `npm test` command:

```
$ npm test
Error: no test specified
```

In this chapter, we'll take a look at all of the script commands npm provides and learn how to organize a project so that collaborators can run its scripts with minimal fuss. We'll also learn how to make certain common npm operations trigger your scripts automatically.

Script commands

There are 4 npm commands for running the scripts with the corresponding names:

- npm test
- npm start
- npm stop
- npm restart

A `package.json` that runs a separate JS file for each of these commands might look like this:

```
...
"scripts": {
  "test": "echo \"Running tests...\" && node test.js",
  "start": "echo \"Starting the server...\" && node start.js",
  "stop": "echo \"Stopping the server...\" && node stop.js",
  "restart": "echo \"Restarting the server...\" && node restart.js"
},
...
```

`"start"` is unique in that it has a default value: It maps to `node server.js` unless overridden.

`npm restart` in flux

It must be mentioned that `npm restart`'s behavior is in flux. Currently, running `npm restart` is equivalent to running `npm stop`, then the `"restart"` script, then `npm start`. This behavior is not consistent with npm's own documentation. It's not clear yet whether the behavior will cbe changed. See https://github.com/isaacs/npm/issues/1999[a] and https://github.com/isaacs/npm/pull/2716[b].

Whatever the outcome of those discussions, you can safely use `npm restart` as shorthand for `npm stop; npm start` if you don't have a `restart` script.

[a] npmissue#1999
[b] npmpullrequest#2716

You can define scripts in your `package.json` with any name you like; they just don't get their own commands. Instead, you'll have to use `npm run-script <script>`. So these two commands are equivalent:

```
$ npm test
Running tests...
Tests passed!
$ npm run-script test
Running tests...
Tests passed!
```

"pre" and "post" scripts

You can define a script that automatically runs before or after another script by giving it the other script's name, prefixed by either "pre" or "post". For example:

```
...
"scripts": {
  "pretest": "echo \"PRE: Tests are about to run.\"",
  "test": "echo \"Running tests...\" && node test.js",
  "posttest": "echo \"POST: Tests have run.\""
},
...
```

With these scripts defined, running `npm test` (or `npm run-script test`) will run all three scripts in the expected order:

```
$ npm test
PRE: Tests are about to run.
Running tests...
Tests passed!
POST: Tests have run.
```

Note that if you return a non-zero exit code from any of the three scripts, the remaining scripts will not run. That makes "pre" scripts a handy way to check preconditions for running the main script.

The script environment

Shell scripting is beyond the scope of this book, but you don't need to be a `bash` expert to write expert-looking `package.json` scripts: They usually just launch a program that lives elsewhere, whether it's a Node module, a shell script, or a binary. Of course, the context from which that program is launched matters.

The npm script user

There are two configs that determine who the user is when you run an npm script: `unsafe-perm` and `user`. If `unsafe-perm` is `true`, then npm runs the script as whoever is running npm, even if they're the root. If `unsafe-perm` is `false`, then npm attempts to run the script as whoever the `user` config is set to. By default, this is `nobody`. (This default, however, may not always apply: See npm issue #3778[28].)

Now here's the tricky part: The default value of `unsafe-perm` is `false` for the root and `true` for everyone else. So unless you override it with `npm set unsafe-perm <value>`, npm will happily let you run scripts as yourself, but will run them as `nobody` (or whatever your `user` config is) if you `sudo`.

If you're every unsure who the user is when a script runs, replace the script with:

[28]https://github.com/isaacs/npm/issues/3778

```
"echo $(whoami)"
```

The PATH

npm ensures that all binaries from the project's dependencies are on the `PATH`, in addition to (but taking precedence over) everything else that was on the `PATH` when you ran the script.

This is extremely handy. Let's say that you use Mocha[29] as your project's test framework. Because Mocha has a binary that takes care of all of the work of finding and running the project's tests, you get to write the simplest test script ever:

```
...
"scripts": {
  "test": "mocha",
},
...
```

For this to work, all you need to do is add Mocha to your project's `devDependencies`. When your collaborators run `npm install`, they'll get the correct version of Mocha in the project's `node_modules`. Additionally, npm automatically copies the `mocha` binary to `node_modules/.bin`, and prefixes your `PATH` with that value when you run `npm test` (or `npm run-script test`).

Script triggers

If you define a script named `"publish"`, `"update"`, or `"install"` npm will run it automatically when that action is performed on the package. You can (and generally should) use the `"pre"` or `"post"` prefix when defining one of these scripts, to ensure that the order is clear.

`publish` scripts are the most common. If your project needs to be compiled from CoffeeScript in a `src` directory to JavaScript in a `lib` directory, for example, you can ensure that compilation always takes place like so:

[29] http://visionmedia.github.io/mocha/

```
...
"scripts": {
  "prepublish": "coffee -co src lib",
},
"devDependencies": {
  "coffee-script": "1.6.2"
}
...
```

`update` and `install` scripts are discouraged by the maintainers of npm. They used to be common when platform-specific compilation was needed. Today, if your project needs a binary to be compiled when it's installed, you should put a `.gyp` ("Generate Your Projects") file in the root of your project instead. npm will run it automatically when your project is installed. For a tutorial on the `.gyp` format, see: https://github.com/springmeyer/hello-gyp[30]

Conclusion

We've seen how adding a few simple scripts to your `package.json` can greatly simplify the lives of package maintainers and users alike. Once you start using npm to manage your project's little development tasks, you'll wonder how you ever lived without it.

[30]https://github.com/springmeyer/hello-gyp