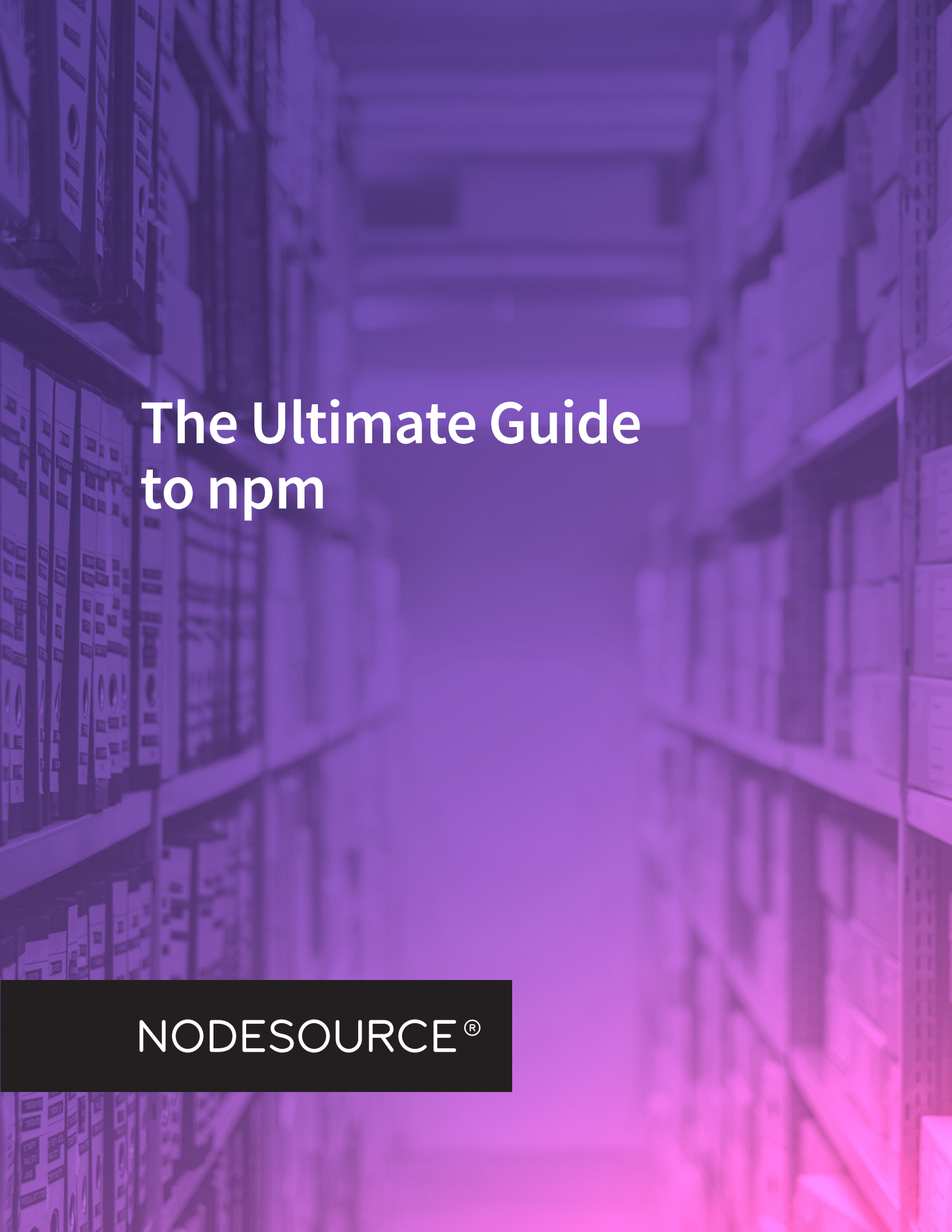


The Ultimate Guide to npm

NODESOURCE®

Contents

Knowing your tools is important; if you can't use the tools at your disposal, you're going to end up spending more time struggling and less time creating, building, and deploying. The Node.js ecosystem has done an absolutely fantastic job in providing tools that help speed development workflows and streamline the process of writing and shipping code.

One of the most important tools in the Node.js developer ecosystem is the npm CLI — it's one tool that really enabled Node.js to become what it is today. It still comes bundled with the core Node.js project, and is an instrumental part of developer workflows.

Knowing this, understanding the npm CLI is critically important to any team working with Node.js today — and is why we've written this Ultimate Guide to npm.

In this guide, we'll cover what you need to know to know to use the npm CLI as a Node.js developer. Topics covered include:

- The basics of the npm CLI
- A deep-dive into the standardized `package.json` file
- Tips and tricks for using npm effectively

The Basics: Getting started with npm

Up and running with the primary tool for the world's largest module ecosystem

Today, npm is a cornerstone of modern web development, whether used exclusively with Node.js as a package manager or as a build tool for the front end.

Really understanding npm as a tool—understanding the core concepts—can be something that's difficult for a beginner. As such, I've written up a basic and **detailed** guide for understanding npm, for those who are entirely new to Node.js, npm, and the surrounding ecosystem.

The Essential npm Commands

When using npm, you're most likely going to be using the command line tool for the majority of your interactions. As such, here's a detailed rundown of the commands you'll encounter and need to use most frequently.

USING `npm init` TO INITIALIZE A PROJECT

The `npm init` command is a step-by-step tool to build out the scaffolding for your project. It will prompt you for input for a few aspects of the project in the following order:

- The project's name
- The project's initial version
- The project's description
- The project's entry point (meaning the project's main file)
- The project's test command (to trigger testing with something like Standard)
- The project's git repository (where the project source can be found)

- The project's keywords (tags related to the project)
- The project's license (this defaults to ISC — most open-source Node.js projects are MIT)

It's worth noting that if you're content with the *suggestion* that the `npm init` command provides next to the prompt, you can simply hit `Return` or `Enter` to accept the suggestion and move on to the next prompt.

Once you run through the `npm init` steps above, a `package.json` file will be generated and placed in the current directory. If you run it in a directory that's not exclusively for your project, don't worry! Generating a `package.json` doesn't really *do* anything, other than create a `package.json` file.

You can either move the `package.json` file to a directory that's dedicated to your project, *or* you can create an entirely new one in such a directory.

```
$ npm init # This will trigger the initialization
```

USING `npm init --yes` TO INSTANTLY INITIALIZE A PROJECT

If you want to get on to building your project, and don't want to spend the (albeit brief) time answering the prompts that come from `npm init`, you can use the `--yes` flag on the `npm init` command to automatically populate all options with the default `npm init` values.

> **Note**: *You can configure what these default values are with the npm configuration commands, which we cover in the section* "Automating npm init Just a Bit More"

```
$ npm init --yes # This will trigger automatically populated
initialization.
```

INSTALL MODULES WITH `npm install`

Installing modules from npm is one of the most basic things you should learn to do when getting started with npm. As you dive deeper, you'll begin to learn some variations on installing modules, but here's the very core of what you need to know to install a standalone module into the current directory:

```
$ npm install <module>
```

In the above command, you'd replace `<module>` with the name of the module you want to install. For example, if you want to install Express (the most used and most well known Node.js web framework), you could run the following command:

```
$ npm install express
```

The above command will install the `express` module into `/node_modules` in the current directory. Whenever you install a module from npm, it will be installed into the `node_modules` folder.

In addition to triggering an install of a single module, you can actually trigger the installation of all modules that are listed as `dependencies` and `devDependencies` in the `package.json` in the current directory. To do so, you'll simply need to run the command itself:

```
$ npm install
```

Once you run this, npm will begin the installation process of all of the current project's dependencies.

As an aside, one thing to note is that there's an alias for `npm install` that you may see in the wild when working with modules from the ecosystem. The alias is `npm i`, where `i` takes the place of `install`.

This seemingly minor alias is a small gotcha for beginners — including me when I was learning — to the Node.js and npm ecosystems, as there's not a standardized, single way that module creators and maintainers will instruct on how to install their module.

This seemingly minor alias is a small gotcha for beginners — including me when I was learning — to the Node.js and npm ecosystems, as there's not a standardized, single way that module creators and maintainers will instruct on how to install their module.

Usage:

```
$ npm install <module> # Where <module> is the name of the module
you want to install

$ npm i <module> # Where <module> is the name of the module you
want to install - using the i alias for installation
```

INSTALL MODULES AND SAVE THEM TO YOUR `package.json` AS A DEPENDENCY

As with `npm init`, the `npm install` command has a flag or two that you'll find useful in your workflow — it'll save you time and effort with regard to your project's `package.json` file.

When you're running `npm install` to install a module, you can add the optional flag `--save` to the command. This flag will add the module as a dependency of your project to the project's `package.json` as an entry in `dependencies`.

Usage:

```
$ npm install <module> --save # Where <module> is the name of the
module you want to install
```

INSTALL MODULES AND SAVE THEM TO YOUR `package.json` AS A DEVELOPER DEPENDENCY

There's a flag that is nearly an exact duplicate, in terms of functionality, of the `--save` flag when installing a module: `--save-dev`. There are a few a key differences between the two — instead of saving the module being installed and added to `package.json` as an entry in `dependencies`, it will save it as an entry in the `devDependencies`.

The semantic difference here is that `dependencies` are for use in production — whatever that would entail for your project. On the other hand, `devDependencies` are a collection of the dependencies that are used in *development* of your application — the modules that you use to build it, but don't need to use when it's *running*. This could include things like testing tools, a local server to speed up your development, and more.

Usage:

```
$ npm install <module> --save-dev # Where <module> is the name of the
module you want to install
```

INSTALL MODULES GLOBALLY ON YOUR SYSTEM

The final, and most common, flag for `npm install` that you should know are the flags to install a module globally on your system.

Global modules can be extremely useful — there are numerous tools, utilities, and more for both development and general usage that you can globally install for use.

To install a module from npm globally, you'll simply need to use the `--global` flag when running the install command to have the module install globally, rather than locally (to the current directory).

> ***Note:*** *One caveat with global modules is that, by default, npm will install them to a system directory, not a local one. With this as the default, you'll need to authenticate as a privileged user on your system to install global modules. As a best practice, you should change the default installation location from a system directory to a user directory. If you'd like to learn to do this, take a peek at* "Tips and Tricks: Working with npm"

Usage:

```
$ npm install <module> --global # Where <module> is the name of
the module you want to install globally

$ npm install <module> -g # Where <module> is the name of the
module you want to install globally, using the -g alias
```

The Basics of Package.json

In this chapter, we'll give you a kickstart introduction to effectively using `package.json` with Node.js and npm.

The `package.json` file is core to the Node.js ecosystem and is a basic part of understanding and working with Node.js, npm, and even modern JavaScript. The `package.json` file is used as a manifest, storing information about applications, modules, packages, and more.

Because understanding `package.json` is essential to working with Node.js, it's a good idea to have a grasp on the commonly found and most important properties of a `package.json` file that you'll need to use `package.json` effectively.

Identifying Metadata Inside `package.json`

THE `NAME` PROPERTY

The `name` property in a `package.json` file is one of the fundamental components of the `package.json` structure. At its core, `name` is a string that is exactly what you would expect — the name of the module that the `package.json` is describing.

Inside your `package.json`, the `name` property as a string would look something like this:

```
"name": "metaverse"
```

There are only a few material restrictions on the `name` property:

- Maximum length of 214 URL-friendly characters
- no uppercase letters
- no leading periods (`.`) or underscores (`_`)

However, some software ecosystems have developed standard naming conventions that enable discoverability.

A few examples of this kind of namespacing are babel-plugin- for Babel and the webpack -loader tooling.

THE `version` PROPERTY

The `version` property is a key part of a `package.json`, as it denotes the current version of the module that the `package.json` file is describing.

While the `version` property isn't required to follow semver (semantic versioning) standards, which is the model used by the vast majority of modules and projects in the Node.js ecosystem — and the module version, according to semver, is what you'll typically find in the `version` property of a `package.json` file.

Inside your `package.json`, the `version` property as a string using semver could look like this:

```
"version": "5.12.4"
```

THE `license` PROPERTY

The `license` property of a `package.json` file is used to note what license the module that the `package.json` file is describing. While there are some complex ways you can use the `license` property of a `package.json` file (to do things like dual-licensing or defining your own license), the most typical usage of it is to use a SPDX License identifier – some examples that you may recognize are `MIT`, `ISC`, and `GPL-3.0`.

Inside your `package.json`, the `license` property with an `MIT` license look like this:

```
"license": "MIT"
```

THE `description` PROPERTY

The `description` property of a `package.json` file is a string that contains a human-readable description about the module – basically, it's the module developer's chance to quickly let users know what *exactly* a module does. The `description` property is frequently indexed by search tools like npm search and the npm CLI search tool to help find relevant packages based on a search query.

Inside your `package.json`, the `description` property would look like this:

```
"description": "The Metaverse virtual reality. The final outcome
of all virtual worlds, augmented reality, and the Internet."
```

THE `keywords` PROPERTY

The `keywords` property inside a `package.json` file is, as you may have guessed, a collection of keywords that describe a module. Keywords can help identify a package, related modules and software, and concepts.

The `keywords` property is always an array, with one or more strings as the array's values – each one of these strings will, in turn, be one of the project's keywords.

Inside your `package.json`, the `keywords` array would look something like this:

```
"keywords": [
      "metaverse",
      "virtual reality",
       "augmented reality",
       "snow crash"
]
```

Functional Metadata Inside `package.json`

THE `main` PROPERTY

The `main` property of a `package.json` is a direction to the entry point to the module that the `package.json` is describing. In a Node.js application, when the module is called via a require statement, the module's exports from the file named in the `main` property will be what's returned to the Node.js application.

Inside your `package.json`, the `main` property, with an entry point of `app.js`, would look like this:

```
"main": "app.js"
```

THE `repository` PROPERTY

The `repository` property of a `package.json` is an array that defines *where* the source code for the module lives. Typically, for open source projects, this would be a public GitHub repo, with the `repository` array noting that the type of version control is `git`, and the URL of the repo itself. One thing to note about this is that it's not just a URL the repo can be accessed from, but the full URL that the version control can be accessed from.

Inside your package.json, the `repository` property would look like this:

```
"repository": {
      "type": "git",
      "url": "https://github.com/bnb/metaverse.git"
}
```

THE `scripts` PROPERTY

The `scripts` property of a `package.json` file is simple conceptually, but is complex functionally, to the point that it's used as a build tool by many.

At its simplest, the `scripts` property contains a set of entries; the key for each entry is a script name, and the corresponding value is a user-defined command to be executed. Scripts are frequently used for testing, building, and streamlining of the needed commands to work with a module.

Inside your package.json, the `scripts` property with a `build` command to execute `node app.js` (presumably to build your application) and a `test` command using Standard would look like this:

```
"scripts": {
      "build": "node app.js",
      "test": "standard”
}
```

To actually run scripts in the `scripts` property of a `package.json`, you'll need to use the default `npm run` command. So, to run the above example's test suite, you'd need to run this:

Usage:

```
$ npm run build
```

That said, to run the test suite with Standard, you'd need to run this:

Usage:

```
$ npm test
```

By default, npm does not require the `run` keyword as part of the given `script` command – `test`, `start`, and `stop` are a few examples of this.

THE `dependencies` PROPERTY

The `dependencies` property of a module's `package.json` is where dependencies — the *other* modules that *this* module uses — are defined. Each entry in the `dependencies` property includes the name and version of other packages required to run this package.

> ***Note:*** *You'll frequently find carets (`^`) and tildes (`~`) included with package versions. These are the notation for version range — taking a deep-dive into these is outside the scope of this guide, but you can learn more in our primer on semver.*

Inside your `package.json`, the `dependencies` property of your module may look something like this:

```
"dependencies": {
      "async": "^0.2.10",
      "npm2es": "~0.4.2",
      "optimist": "~0.6.0",
      "request": "~2.30.0",
      "skateboard": "^1.5.1",
      "split": "^0.3.0",
      "weld": "^0.2.2"
},
```

THE `devDependencies` PROPERTY

The `devDependencies` property of `package.json` is almost identical to the `dependencies` property in terms of structure. The main difference: while the `dependencies` property is used to define the dependencies that a module needs to run in *production*, `devDependencies` property is usually used to define the dependencies the module needs to run in *development*.

Inside your `package.json`, the `devDependencies` property would look something like this:

```
"devDependencies": {
      "escape-html": "^1.0.3",
      "lucene-query-parser": "^1.0.1"
}
```

Understanding `dependencies` and `devDependencies` in `package.json`

The other majorly important aspect of `package.json` is that it contains a collection of any given project's dependencies. These dependencies are the modules that the project relies on to function properly.

Having dependencies in your project's `package.json` allows the project to install the versions of the modules it depends on. By running an install command inside of a project, you can install *all* of the dependencies that are listed in the project's `package.json` – meaning they don't have to be (and almost never should be) bundled with the project itself.

Second, it allows the separation of dependencies that are needed for production and dependencies that are needed for development. In production, you're likely not going to need a tool to watch your CSS files for changes and refresh the app when they change. But in both production and development, you'll want to have the modules that enable what you're trying to accomplish with your project – things like your web framework, API tools, and code utilities.

What would a project's `package.json` look like with `dependencies` and `devDependencies`? Let's expand on the previous example of a `package.json` to include some.

```
{
      "name": "metaverse",
      "version": "0.92.12",
      "description": "The Metaverse virtual reality. The final
      outcome of all virtual worlds, augmented reality, and the
      Internet.",
      "main": "index.js",
      "license": "MIT",
      "devDependencies": {
            "mocha": "~3.1",
            "native-hello-world": "^1.0.0",
            "should": "~3.3",
            "sinon": "~1.9"
      },
      "dependencies": {
            "fill-keys": "^1.0.2",
            "module-not-found-error": "^1.0.0",
            "resolve": "~1.1.7"
      }
}
```

One key difference between the dependencies and the other common parts of `package.json` is that they're both objects, with multiple key/value pairs. Every key in both `dependencies` and `devDependencies` is a name of a package, and every value is the version range that's acceptable to install (according to semver).

Tips and Tricks: Working with npm

Quick and simple tricks to boost productivity by using the npm CLI

Once you've mastered the basics of working with npm, there are dozens of built-in features that can help you manage Node.js projects more effectively.

It can be a little daunting to try to memorize all of these features at once, so we recommend starting with this list of time-saving tricks — you'll likely find that at least a few will be applicable to almost every project you work on.

Getting More Information for Individual Packages

OPEN A PACKAGE'S HOMEPAGE

Usage

```
$ npm home $package
```

Running the `home` command will open the homepage of the package you're running it against. Running against the `lodash` package will bring you to the Lodash website. This command can run without needing to have the package installed globally on your machine or within the current project.

OPEN PACKAGE'S GITHUB REPO

Usage:

```
$ npm repo $package
```

Similar to `home`, the `repo` command will open the GitHub repository of the package you're running it against. Running against the `express` package will bring you to the official Express repo. Also like `home`, you don't need to have the package installed.

CHECK A PACKAGE FOR OUTDATED DEPENDENCIES

```
⬢ bnb.im [master] npm outdated
Package           Current  Wanted  Latest  Location
metalsmith-prism    2.2.0   2.2.0   3.0.0  bnb.im
⬢ bnb.im [master]
```

Usage:

```
$ npm outdated
```

You can run the `outdated` command within a project, and it will check the npm registry to see if any of your packages are outdated. It will print out a list in your command line of the current version, the wanted version, and the latest version.

Managing Your Node.js Projects

VIEW GLOBALLY INSTALLED NODE MODULES

I've run into a problem a handful of times where I've globally installed a module that I want to start using when I'm working. When it finally comes time to use it, I can't actually remember what it was.

Thankfully, there's a super easy way to solve this – you can list out all your globally installed modules with one simple command:

Usage:

```
$ npm ls -g --depth 0
```

The command will run for a bit, depending on how many global modules you have, and then print out a list of them all.

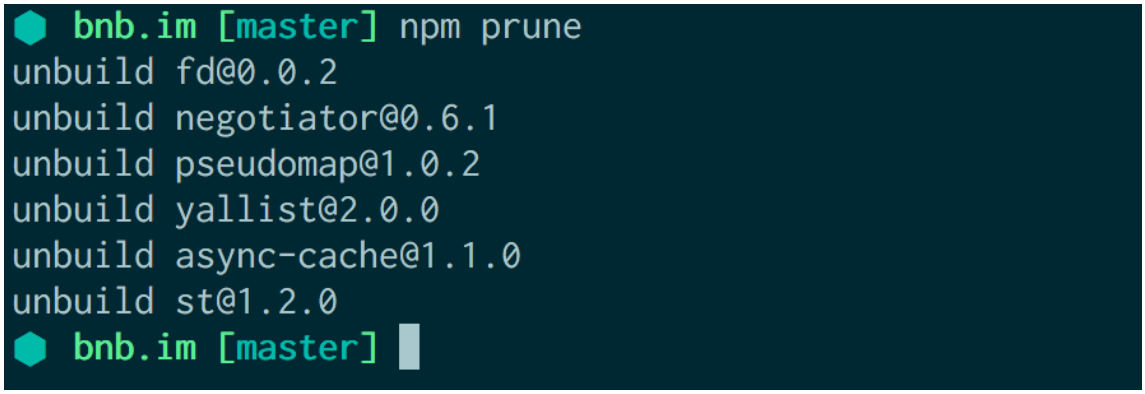
CHECK FOR PACKAGES NOT DECLARED IN `package.json`

Usage:

```
$ npm prune
```

When you run `prune`, the npm CLI will run through your `package.json` and compare it to your project's `/node_modules` directory. It will print a list of modules that aren't in your `package.json`.

The `npm prune` command then strips out those packages, and removes any you haven't manually added to `package.json` or that were `npm install`-ed without using the `--save` flag.



Speed Up with Workflow Enhancements

ADDING NPM COMMAND AUTOCOMPLETION TO YOUR SHELL

If you want to get a quick improvement to your npm productivity, you can add autocompletion for npm to your shell with just one command.

For `bash`, you can add npm autocompletion with:

Usage:

```
$ npm completion >> ~/.bashrc
```

For `zsh`, you can add npm autocompletion with:

Usage:

```
$ npm completion >> ~/.zshrc
```

And now you'll have tab autocomplete for npm commands.

Dealing with Dependencies

LOCK DOWN YOUR DEPENDENCIES VERSIONS

Usage:

```
$ npm shrinkwrap
```

Using `shrinkwrap` in your project generates an `npm-shrinkwrap.json` file. This allows you to pin the dependencies of your project to the specific version you're currently using within your `node_modules` directory. When you run `npm install` and there is a `npm-shrinkwrap.json` present, it will override the listed dependencies and any semver ranges in `package.json`.

If you need verified consistency across `package.json`, `npm-shrinkwrap.json` and `node_modules` for your project, you should consider using npm-shrinkwrap.

```
bnb.im [master] npm shrinkwrap
npm WARN shrinkwrap Excluding devDependency: gh-pages@0.11.0 { handlebars: '^4.0.5',
npm WARN shrinkwrap   metalsmith: '^2.1.0',
npm WARN shrinkwrap   'metalsmith-assets': '^0.1.0',
npm WARN shrinkwrap   'metalsmith-collections': '^0.7.0',
npm WARN shrinkwrap   'metalsmith-filemetadata': '^1.0.0',
npm WARN shrinkwrap   'metalsmith-if': '^0.1.1',
npm WARN shrinkwrap   'metalsmith-layouts': '^1.4.2',
npm WARN shrinkwrap   'metalsmith-markdown': '^0.2.1',
npm WARN shrinkwrap   'metalsmith-prism': '^2.1.1',
npm WARN shrinkwrap   'metalsmith-validate': '^0.1.4',
npm WARN shrinkwrap   'metalsmith-word-count': '0.0.4' }
wrote npm-shrinkwrap.json
bnb.im [master]
```

STRIP YOUR PROJECT'S `devDependencies` FOR A PRODUCTION ENVIRONMENT

When your project is ready for production, make sure you install your packages with the added `--production` flag. The `--production` flag installs your `dependencies`, ignoring your `devDependencies`. This ensures that your development tooling and packages won't go into the production environment.

Additionally, you can set your `NODE_ENV` environment variable to `production` to ensure that your project's `devDependencies` are never installed.

LINK YOUR LOCAL DEPENDENCIES FOR DEVELOPMENT

Usage:

```
$ npm link
```

If you're working with a package that has out-of-date dependencies, npm's `link` can help. It can allow you to link a a newer, local copy of the out-of-date dependency and test your module with the newer, local version. This local development step provides a major assist with verifying code changes are valid before actually publishing to npm.

If the package is very deep in the dependency tree, or is depended on multiple times, you may want to look at a tool like lnr.

For example, let's say we want to update the `cookie` module and verify it works with `express` first.

```
$ cd cookie  // Go to your local cookie package

$ npm link  // Link the local cookie package

$ cd ../express  // Go to your application

$ npm link cookie  // Links the local cookie package to your
application

$ npm install  // Will ignore the local cookie package when
installing
modules
```

This allows us to make our changes to the code module and update express at the same time if need be. We can then publish our local `cookie` package and update our app, `express`, with the new package version.

Package Security

USE `.npmignore` WITH CAUTION

If you haven't been using `.npmignore`, it uses `.gitignore` as a fallback, with a few additional sane defaults.

What many don't realize that once you add a `.npmignore` file to your project the `.gitignore` rules are (ironically) ignored. The result is you will need to audit the two ignore files in sync to prevent leaks of sensitive information when publishing.

ENSURE YOUR PACKAGES ARE RUNNING IN NODE SECURELY

If you're running a specific Node version in production, you need to make sure your dependencies actually *work* with the Node version that you're running. Luckily, there's a quick trick to get npm to verify that your packages *say* they're compatible with the version of Node you're running.

Usage:

```
$ npm config set engine-strict true
```

Additionally, if you want to block npm scripts (for security reasons), you can set the `ignore-script` config option – this will *completely* block any scripts in an application's `package.json` – including the application's dependencies.

To do this, you can run:

Usage:

```
$ npm config set ignore-scripts
```

Automating `npm init`

When you're creating a new module from scratch, you'll typically start out with the `npm init` command. One thing that some developers don't know is that you can actually automate this process significantly with a few choice `npm config set ...` commands that set default values for the `npm init` prompts.

You can easily set your name, email, URL, license, and initial module version with a few commands:

Usage:

```
$ npm config set init.author.name "Hiro Protagonist"
$ npm config set init.author.email "hiro@showcrash.io"
$ npm config set init.author.url "http://hiro.snowcrash.io"
$ npm config set init.license "MIT"
$ npm config set init.version "0.0.1"
```

In the above example, I've set up some defaults for Hiro. This personal information won't change too frequently, so setting up some defaults is helpful and allows you to skip over entering the same information in manually every time.

Additionally, the above commands set up two defaults that are related to your module.

The first default is the initial license that will be automatically suggested by the `npm init` command. I personally like to default to `MIT`, and much of the rest of the Node.js ecosystem does the same. That said, you can set this to whatever you'd like – it's a nice optimization to just be able to nearly automatically select your license of choice.

The second default is the initial version. This is actually one that made me happy, as whenever I tried building out a module I never wanted it to start out at version `1.0.0`, which is what `npm init` defaults to. I personally set it to `0.0.1` and then increment the version as I go with the `npm version [ major | minor | patch ]` command.

If you'd like to completely customize your initialization script, you can point to a self-made default init script by running the following command:

Usage:

```
$ npm config set init-module ~/.npm-init.js
```

Here's a sample script that prompts for private settings and creates a GitHub repo if you want. Make sure you change the default GitHub username (`YOUR_GITHUB_USERNAME`) as the fallback for the GitHub username environment variable.

```
var cp = require('child_process');
var priv;
var USER = process.env.GITHUB_USERNAME || 'YOUR_GITHUB_USERNAME';

module.exports = {

  name: prompt('name', basename || package.name),
  version: '0.0.1',
  private: prompt('private', 'true', function(val){
       return priv = (typeof val === 'boolean') ? val : !!val.
match('true')
  }),
  create: prompt('create github repo', 'yes', function(val){
       val = val.indexOf('y') !== -1 ? true : false;
       if(val){
            console.log('enter github password:');
            cp.execSync("curl -u '"+USER+"'
            https://api.github.com/user/repos -d " + "'{\"name\":
            \""+basename+"\", \"private\": "+ ((priv) ? 'true' :
'false')
            +"}' ");
            cp.execSync('git remote add origin '+
            'https://github.com/'+USER+'/' + basename + '.git');
       }
       return undefined;
  }),
```

```
  main: prompt('entry point', 'index.js',
  repository: {
       type: 'git',
       url: 'git://github.com/'+USER+'/' + basename + '.git'
  },
  bugs: {
       url: 'https://github.com/'+USER'/' + basename + 'issues'
  },
  homepage: "https://github.com/"+USER+"/" + basename,
  keywords: prompt(function (s) {
       return s.split(/\s+/)
  }),
  license: 'MIT',
  cleanup: function(cb){
       cb(null, undefined)
  }
}
```

Change Your npm Registry

More options for registries have become available as the Node.js ecosystem has grown. You may want to set your registry to a cache of the modules you know you need for your apps. Or, you may be using Certified Modules as a custom npm registry. There's even a separate registry for Yarn, a topic that is both awesome and totally out of scope for this guide.

So, if you'd like to set a custom registry, you can run a pretty simple one-line command:

Usage:

```
$ npm config set registry "https://my.registry.nodesource.io/"
```

In this example, I've set the registry URL to an example of a Certified Modules registry – that said, the exact URL in the command can be replaced with any registry that's compatible. To reset your registry back to the default npm registry, you can simply run the same command pointing to the standard registry:

Usage:

```
$ npm config set registry "https://registry.npmjs.com/"
```

Changing the console output of `npm install` with `loglevel`

When you `npm install`, a variety of information is available to you. By default, the npm CLI limits how much of this information is actually output into the console when installing. There are varying degrees of output that you can assign at install, or by default, if you change it with `npm config` in your `.npmrc` file. The options, from least to most output, are: `silent`, `error`, `warn`, `http`, `info`, `verbose`, and `silly`.

If you'd like to get a bit more information (or a bit less, depending on your preferences) when you `npm install`, you can change the default loglevel.

Usage:

```
$ npm config set loglevel="http"
```

If you tinker around with this config a bit and would like to reset to what the npm CLI *currently* defaults to, you can run the above command with `warn` as the loglevel:

Usage:

```
$ npm config set loglevel="warn"
```

Change the Install Location for Global Modules

This is a really awesome change — it has a few steps, but is really worth it. With a few commands, you can change where the npm CLI installs global modules by default. Normally, it installs them to a privileged system folder – this requires administrative access, meaning that a global install requires `sudo` access on UNIX-based systems.

If you change the default global prefix for `npm` to an unprivileged directory, for example `~/.global-modules`, you'll not need to authenticate when you install a global module. That's one benefit – another is that globally installed modules won't be in a system directory, reducing the likelihood of a malicious module (intentionally or not) doing something you didn't want it to on your system.

To get started, we're going to create a new folder called `global-modules` and set the npm prefix to it:

Usage:

```
$ mkdir ~/.global-modules
$ npm config set prefix "~/.global-modules"
```

Next, if you don't already have a file called `~/.profile`, create one in your root user directory. Now, add the following line to the `~/.profile` file:

Usage:

```
$ export PATH=~/.global-modules/bin:$PATH
```

Adding that line to the `~/.profile` file will add the `global-modules` directory to your PATH, and enable you to use it for npm global modules.

Now, flip back over to your terminal and run the following command to update the PATH with the newly updated file:

Usage:

```
$ source ~/.profile
```

What next?

The ecosystem surrounding npm is continuously evolving. There are always going to be hard problems to solve – things like security, licensing, management, module optimization, and more. Tools like NodeSource Certified Modules are aiming to directly help alleviate these problems in a secure and reliable way. That said, there will never be *one right answer* to any individual's or organization's direct needs, but rather many answers to solve the problems – that we face both now and in the future – on a case-by-case basis.

If you've got any specific questions you'd like to ask, feel free to reach out to us at @NodeSource on Twitter – we'd love to hear your thoughts and feedback on The Ultimate Guide to npm.