



## Getting Started

- [01 - What is npm?](#)
- [02 - Installing Node.js and updating npm](#)
- [03 - Fixing npm permissions](#)
- [04 - Installing npm packages locally](#)
- [05 - Using a `package.json`](#)
- [06 - Updating local packages](#)
- [07 - Uninstalling local packages](#)
- [08 - Installing npm packages globally](#)
- [09 - Updating global packages](#)
- [10 - Uninstalling global packages](#)
- [11 - Creating Node.js modules](#)
- [12 - Publishing npm packages](#)
- [13 - Semantic versioning and npm](#)
- [14 - Working with scoped packages](#)
- [15 - Using tags](#)

## How npm works

- [01 - Packages](#)
- [02 - npm v2](#)
- [03 - npm v3](#)
- [04 - npm v3 Duplication](#)
- [05 - npm v3 Non-determinism](#)

## Private Modules

- [01 - Working with private modules](#)
- [02 - Downloading modules to CI/deployment servers](#)
- [03 - Docker and private modules](#)

## Organizations

- [01 - What are Organizations?](#)
- [02 - Setup](#)
- [03 - Roles](#)
- [04 - The Developers Team](#)
- [05 - Teams](#)
- [06 - Sponsorship](#)
- [07 - Scoping New Packages](#)
- [08 - Package Access](#)
- [09 - Pre-Existing Packages](#)

## Using npm

- [coding-style](#)
- [config](#)
- [developers](#)
- [disputes](#)
- [orgs](#)
- [registry](#)
- [removing-npm](#)
- [scope](#)
- [scripts](#)
- [semver](#)

## CLI Commands

- [access](#)
- [adduser](#)
- [bin](#)
- [bugs](#)
- [build](#)
- [bundle](#)
- [cache](#)
- [completion](#)
- [config](#)
- [dedupe](#)
- [deprecate](#)
- [dist-tag](#)

[docs](#)  
[edit](#)  
[explore](#)  
[help](#)  
[help-search](#)  
[init](#)  
[install](#)  
[install-test](#)  
[link](#)  
[logout](#)  
[ls](#)  
[npm](#)  
[outdated](#)  
[owner](#)  
[pack](#)  
[ping](#)  
[prefix](#)  
[prune](#)  
[publish](#)  
[rebuild](#)  
[repo](#)  
[restart](#)  
[root](#)  
[run-script](#)  
[search](#)  
[shrinkwrap](#)  
[star](#)  
[stars](#)  
[start](#)  
[stop](#)  
[tag](#)  
[team](#)  
[test](#)  
[uninstall](#)  
[unpublish](#)  
[update](#)  
[version](#)  
[view](#)  
[whoami](#)

## Configuring npm

[folders](#)  
[npmrc](#)  
[package.json](#)

## npm policy documents

[conduct](#)  
[disputes](#)  
[dmca](#)  
[license](#)  
[open-source-terms](#)  
[organization-plan](#)  
[personal-plan](#)  
[privacy](#)  
[private-terms](#)  
[README](#)  
[receiving-reports](#)  
[recruiting-process](#)  
[security](#)  
[terms](#)  
[trademark](#)

## npm, the Company

[about](#)  
[jobs](#)  
[private-npm](#)  
[security](#)  
[weekly](#)

## All Docs

## What is npm?

npm makes it easy for JavaScript developers to share and reuse code, and it makes it easy to update the code that you're sharing.

If you've been working with Javascript for a while, you might have heard of npm: npm makes it easy for Javascript developers to share the code that they've created to solve particular problems, and for other developers to reuse that code in their own applications.

Once you're depending on this code from other developers, npm makes it really easy to check to see if they've made any updates to it, and to download those updates when they're made.

These bits of reusable code are called packages, or sometimes modules. A package is just a directory with one or more files in it, that also has a file called "package.json" with some metadata about this package. A typical application, such as a website, will depend on dozens or hundreds of packages. These packages are often small. The general idea is that you create a small building block which solves one problem and solves it well. This makes it possible for you to compose larger, custom solutions out of these small, shared building blocks.

There's lots of benefits to this. It makes it possible for your team to draw on expertise outside of your organization by bringing in packages from people who have focused on particular problem areas. But even if you don't reuse code from people outside of your organization, using this kind of module based approach can actually help your team work together better, and can also make it possible to reuse code across projects.

You can find packages to help you build your application by browsing the npm website. When you're browsing the website, you'll find different kinds of packages. You'll find lots of node modules. npm started as the node package manager, so you'll find lots of modules which can be used on the server side. There are also lots of packages which add commands for you to use in the command line. And at this point you can find a number of packages which can be used in the browser, on the front end.

So now that you have an idea of what npm can do, let's talk about how it works. When people talk about npm, they can be talking about one of three things. They could be talking about the website, which we've just been looking at. Or they could be talking about the registry, which is a big database of information about packages that people are sharing. Or the third thing they could be talking about is the client: when a developer decides to share their code, they use the npm client which is installed on their computer to publish that code up to the registry. And once there's an entry for this package in the registry, then other developers can use their npm clients to install the package from the registry. The entry in the registry for this package is also reflected on the website, where there's a page dedicated to this new package.

So that's what npm is. It's a way to reuse code from other developers, and also a way to share your code with them, and it makes it easy to manage the different versions of code.

---

Last modified November 04, 2016      Found a typo? Send a [pull request!](#)

## Installing Node.js and updating npm

### Installing Node.js

If you're using OS X or Windows, the best way to install Node.js is to use one of the installers from the [Node.js download page](#). If you're using Linux, you can use the installer, or you can check [NodeSource's binary distributions](#) to see whether or not there's a more recent version that works with your system.

Test: Run `node -v` . The version should be higher than v0.10.32.

### Updating npm

Node comes with npm installed so you should have a version of npm. However, npm gets updated more frequently than Node does, so you'll want to make sure it's the latest version.

```
npm install npm@latest -g
```

Test: Run `npm -v` . The version should be higher than 2.1.8.

### Installing npm manually

For more advanced users.

The npm module is available for download at <https://registry.npmjs.org/npm/-/npm-{VERSION}.tgz> .

## Fixing npm permissions

You may receive an **EACCES** error when you try to install a package globally. This indicates that you do not have permission to write to the directories that npm uses to store global packages and commands.

You can fix this problem using one of three options:

1. Change the permission to npm's default directory.
2. Change npm's default directory to another directory.
3. Install node with a package manager that takes care of this for you.

You should back-up your computer before moving forward.

## Option 1: Change the permission to npm's default directory

1. Find the path to npm's directory:

```
npm config get prefix
```

For many systems, this will be `/usr/local`.

**WARNING:** If the displayed path is *just* `/usr`, switch to [Option 2](#) or you will mess up your permissions.

2. Change the owner of npm's directories to the name of the current user (your username!):

```
sudo chown -R $(whoami) $(npm config get prefix){lib/node_modules,bin,share}
```

This changes the permissions of the sub-folders used by npm and some other tools ( `lib/node_modules`, `bin`, and `share` ).

## Option 2: Change npm's default directory to another directory

There are times when you do not want to change ownership of the default directory that npm uses (i.e. `/usr`) as this could cause some problems, for example if you are sharing the system with other users.

Instead, you can configure npm to use a different directory altogether. In our case, this will be a hidden directory in our home folder.

1. Make a directory for global installations:

```
mkdir ~/.npm-global
```

2. Configure npm to use the new directory path:

```
npm config set prefix '~/.npm-global'
```

3. Open or create a `~/.profile` file and add this line:

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

4. Back on the command line, update your system variables:

```
source ~/.profile
```

Test: Download a package globally without using `sudo`.

```
npm install -g jshint
```

Instead of steps 2-4 you can also use the corresponding ENV variable (e.g. if you don't want to modify `~/.profile`):

```
NPM_CONFIG_PREFIX=~/.npm-global
```

## Option 3: Use a package manager that takes care of this for you.

If you're doing a fresh install of node on Mac OS you can avoid this problem altogether by using the [Homebrew](#) package manager. Homebrew sets things up out of the box with the correct permissions.

```
brew install node
```

---

Last modified June 17, 2016

Found a typo? Send a [pull request!](#)

## Installing npm packages locally

There are two ways to install npm packages: locally or globally. You choose which kind of installation to use based on how you want to use the package.

If you want to depend on the package from your own module using something like Node.js' `require`, then you want to install locally, which is `npm install`'s default behavior. On the other hand, if you want to use it as a command line tool, something like the grunt CLI, then you want to [install it globally](#).

To learn more about the `install` command's behavior, check out the [CLI doc page](#).

## Installing

A package can be downloaded with the command

```
> npm install <package_name>
```

This will create the `node_modules` directory in your current directory(if one doesn't exist yet), and will download the package to that directory.

### Test:

To confirm that `npm install` worked correctly, check to see that a `node_modules` directory exists and that it contains a directory for the package(s) you installed. You can do this by running `ls node_modules` on Unix systems, e.g. "OSX", "Debian", or `dir node_modules` on Windows.

### Example:

Install a package called `lodash`. Confirm that it ran successfully by listing the contents of the `node_modules` directory and seeing a directory called `lodash`.

```
> npm install lodash
> ls node_modules           # use `dir` for Windows

#=> lodash
```

## Which version of the package is installed?

If there is no `package.json` file in the local directory, the latest version of the package is installed.

If there is `package.json` file, the latest version satisfying the [semver rule](#) declared in `package.json` for that package (if there is any) is installed.

## Using the installed package

Once the package is in `node_modules`, you can use it in your code. For example, if you are creating a Node.js module, you can `require` it.

### Example:

Create a file named `index.js`, with the following code:

```
// index.js
var lodash = require('lodash');

var output = lodash.without([1, 2, 3], 1);
console.log(output);
```

Run the code using `node index.js`. It should output `[2, 3]`.

If you had not properly installed `lodash`, you would receive this error:

```
module.js:340
  throw err;
      ^
Error: Cannot find module 'lodash'
```

To fix this, run `npm install lodash` in the same directory as your `index.js`.

---

Last modified June 17, 2016

Found a typo? Send a [pull request!](#)

## Using a `package.json`

The best way to manage locally installed npm packages is to create a `package.json` file.

A `package.json` file affords you a lot of great things:

1. It serves as documentation for what packages your project depends on.
2. It allows you to specify the versions of a package that your project can use using [semantic versioning rules](#).
3. Makes your build reproducible which means that its way easier to share with other developers.

## Requirements

As a bare minimum, a `package.json` must have:

- `"name"`
  - all lowercase
  - one word, no spaces
  - dashes and underscores allowed
- `"version"`
  - in the form of `x.x.x`
  - follows [semver spec](#)

For example:

```
{
  "name": "my-awesome-package",
  "version": "1.0.0"
}
```

## Creating a `package.json`

To create a `package.json` run:

```
> npm init
```

This will initiate a command line questionnaire that will conclude with the creation of a `package.json` in the directory you initiated the command.

### The `--yes` `init` flag

The extended CLI Q&A experience is not for everyone, and often if you are comfortable with using a `package.json` you'd like a more expedited experience.

You can get a default `package.json` by running `npm init` with the `--yes` or `-y` flag:

```
> npm init --yes
```

This will ask you only one question, **author** . Otherwise it will fill in default values:

```
> npm init --yes
Wrote to /home/ag_dubs/my_package/package.json:
```

```
{
  "name": "my_package",
  "version": "1.0.0",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "ag_dubs",
  "license": "ISC",
  "repository": {
    "type": "git",
    "url": "https://github.com/ashleygwilliams/my_package.git"
  },
  "bugs": {
    "url": "https://github.com/ashleygwilliams/my_package/issues"
  },
  "homepage": "https://github.com/ashleygwilliams/my_package"
}
```

- **name** : defaults to author name unless in a **git** directory, in which case it will be the name of the repository
- **version** : always **1.0.0**
- **main** : always **index.js**
- **scripts** : by default creates a empty **test** script
- **keywords** : empty
- **author** : whatever you provided the CLI
- **license** : **ISC**
- **repository** : will pull in info from the current directory, if present
- **bugs** : will pull in info from the current directory, if present
- **homepage** : will pull in info from the current directory, if present

You can also set several config options for the init command. Some useful ones:

```
> npm set init.author.email "wombat@npmjs.com"
> npm set init.author.name "ag_dubs"
> npm set init.license "MIT"
```

#### **NOTE:**

If there is no description field in the **package.json** , npm uses the first line of the **README.md** or **README** instead. The description helps people find your package on npm search, so it's definitely useful to make a custom description in the **package.json** to make your package more discoverable.

## Customizing the **init** process

It is also possible to totally customize the information created and the questions asked during the init process. This is done by creating a custom **.npm-init.js** . By default, npm will look in your home directory. **~/.npm-init.js**

A simple **.npm-init.js** could look something like this:

```
module.exports = {
  customField: 'Custom Field',
  otherCustomField: 'This field is really cool'
}
```

Running **npm init** with this file in your home directory, would output a **package.json** similiar to this:

```
{
  customField: 'Custom Field',
  otherCustomField: 'This field is really cool'
}
```

Customizing the questions is also possible, by using the **prompt** function.

```
module.exports = prompt("what's your favorite flavor of ice cream buddy?", "I LIKE THEM ALL");
```

To learn more on how to create more advanced customizations, checkout the docs for [init-package-json](https://docs.npmjs.com/all/init-package-json)

# Specifying Packages

To specify the packages your project depends on, you need to list the packages you'd like to use in your `package.json` file. There are 2 types of packages you can list:

- `"dependencies"` : these packages are required by your application in production
- `"devDependencies"` : these packages are only needed for development and testing

## Manually editing your `package.json`

You can manually edit your `package.json`. You'll need to create an attribute in the package object called `dependencies` that points to an object. This object will hold attributes named after the packages you'd like to use, that point to a [semver](#) expression that specifies what versions of that project are compatible with your project.

If you have dependencies you only need to use during local development, you will follow the same instructions as above but in an attribute called `devDependencies`.

For example: The project below uses any version of the package `my_dep` that matches major version 1 in production, and requires any version of the package `my_test_framework` that matches major version 3, but only for development:

```
{
  "name": "my_package",
  "version": "1.0.0",
  "dependencies": {
    "my_dep": "^1.0.0"
  },
  "devDependencies": {
    "my_test_framework": "^3.1.0"
  }
}
```

## The `--save` and `--save-dev` install flags

The easier (and more awesome) way to add dependencies to your `package.json` is to do so from the command line, flagging the `npm install` command with either `--save` or `--save-dev`, depending on how you'd like to use that dependency.

To add an entry to your `package.json`'s `dependencies`:

```
npm install <package_name> --save
```

To add an entry to your `package.json`'s `devDependencies`:

```
npm install <package_name> --save-dev
```

# Managing dependency versions

npm uses Semantic Versioning, or, as we often refer to it, SemVer, to manage versions and ranges of versions of packages.

If you have a `package.json` file in your directory and you run `npm install`, then npm will look at the dependencies that are listed in that file and download the latest versions satisfying [semver rules](#) for all of those.

To learn more about semantic versioning, check out our [Getting Started "Semver" page](#).

---

Last modified November 04, 2016

Found a typo? Send a [pull request!](#)

## Updating local packages

Every so often, you should update the packages you depend on so you can get any changes that have been made to code upstream.

To do this, run `npm update` in the same directory as your `package.json` file.

Test: Run `npm outdated`. There should not be any results.



## See Also

- [npm-update](#)
- [npm-outdated](#)

---

Last modified January 12, 2017

Found a typo? Send a [pull request!](#)

## Uninstalling local packages

You can remove a package from your `node_modules` directory using `npm uninstall <package>` :

```
npm uninstall lodash
```

To remove it from the dependencies in `package.json` , you will need to use the `save` flag:

```
npm uninstall --save lodash
```

Note: if you installed the package as a "devDependency" (i.e. with `--save-dev` ) then `--save` won't remove it from `package.json` . You have to use `--save-dev` to uninstall it.

### Test:

To confirm that `npm uninstall` worked correctly, check to see that the `node_modules` directory exists, but that it does not contain a directory for the package(s) you uninstalled. You can do this by running `ls node_modules` on Unix systems, e.g. "OSX", "Debian", or `dir node_modules` on Windows.

### Example:

Install a package called `lodash` . Confirm that it ran successfully by listing the contents of the `node_modules` directory and seeing a directory called `lodash` .

Uninstall `lodash` with `npm uninstall` . Confirm that it ran successfully by listing the contents of the `node_modules` directory and confirming the absence of a directory called `lodash` .

```
> npm install lodash
> ls node_modules          # use `dir` for Windows

#=> lodash

> npm uninstall lodash
> ls node_modules

#=>
```

---

Last modified January 12, 2017

Found a typo? Send a [pull request!](#)

## Installing npm packages globally

There are two ways to install npm packages: locally or globally. You choose which kind of installation to use based on how you want to use the package.

If you want to use it as a command line tool, something like the grunt CLI, then you want to install it globally. On the other hand, if you want to depend on the package from your own module using something like Node's `require` , then you want to install locally.

To download packages globally, you simply use the command `npm install -g <package>` , e.g.:

```
npm install -g jshint
```

If you get an EACCES error, you *should* [fix your permissions](#). You could also try using `sudo` , but this **should be avoided**:

```
sudo npm install -g jshint
```

## Updating global packages

To update global packages, you can use `npm update -g <package>` :

```
npm update -g jshint
```

To find out which packages need to be updated, you can use `npm outdated -g --depth=0` .

To update all global packages, you can use `npm update -g` . However, for npm versions less than 2.6.1, [this script](#) is recommended to update all outdated global packages.

## Uninstalling global packages

Global packages can be uninstalled with `npm uninstall -g <package>` :

```
npm uninstall -g jshint
```

## Creating Node.js modules

Node.js modules are one kind of package which can be published to npm. When you create a new module, you want to start with the `package.json` file.

You can use `npm init` to create the `package.json` . It will prompt you for values for the `package.json` fields. The two required fields are name and version. You'll also want to have a value for main. You can use the default, `index.js` .

If you want to add information for the author field, you can use the following format (email and web site are both optional):

```
Your Name <email@example.com> (http://example.com)
```

Once your `package.json` file is created, you'll want to create the file that will be loaded when your module is required. If you used the default, this is `index.js` .

In that file, add a function as a property of the `exports` object. This will make the function available to other code.

```
exports.printMsg = function() {  
  console.log("This is a message from the demo package");  
}
```

Test:

1. Publish your package to npm
2. Make a new directory outside of your project and cd into it
3. Run `npm install <package>`
4. Create a test.js file which requires the package and calls the method
5. Run `node test.js` . The message should be output.

## Publishing npm packages

You can publish any directory that has a `package.json` file, e.g. a [node module](#).

# Creating a user

To publish, you must have a user on the npm registry. If you don't have one, create it with `npm adduser`. If you created one on the site, use `npm login` to store the credentials on the client.

Test: Use `npm config ls` to ensure that the credentials are stored on your client. Check that it has been added to the registry by going to <https://npmjs.com/~>.

# Publishing the package

Use `npm publish` to publish the package.

Note that everything in the directory will be included unless it is ignored by a local `.gitignore` or `.npmignore` file as described in [npm-developers](#).

Also make sure there isn't already a package with the same name, owned by somebody else.

Test: Go to <https://npmjs.com/package/<package>>. You should see the information for your new package.

# Updating the package

When you make changes, you can update the package using `npm version <update_type>`, where `update_type` is one of the semantic versioning release types, patch, minor, or major. This command will change the version number in `package.json`. Note that this will also add a tag with this release number to your git repository if you have one.

After updating the version number, you can `npm publish` again.

Test: Go to <https://npmjs.com/package/<package>>. The package number should be updated.

The README displayed on the site will not be updated unless a new version of your package is published, so you would need to run `npm version patch` and `npm publish` to have a documentation fix displayed on the site.

---

Last modified July 18, 2016

Found a typo? Send a [pull request!](#)

## Semantic versioning and npm

Semantic versioning is a standard that a lot of projects use to communicate what kinds of changes are in this release. It's important to communicate what kinds of changes are in a release because sometimes those changes will break the code that depends on the package.

## Semver for publishers

If a project is going to be shared with others, it should start at `1.0.0`, though some projects on npm don't follow this rule.

After this, changes should be handled as follows:

- Bug fixes and other minor changes: Patch release, increment the last number, e.g. 1.0.1
- New features which don't break existing features: Minor release, increment the middle number, e.g. 1.1.0
- Changes which break backwards compatibility: Major release, increment the first number, e.g. 2.0.0

## Semver for consumers

As a consumer, you can specify which kinds of updates your app can accept in the `package.json` file.

If you were starting with a package 1.0.4, this is how you would specify the ranges:

- Patch releases: `1.0` or `1.0.x` or `~1.0.4`

- Minor releases: `1` or `1.x` or `^1.0.4`
- Major releases: `*` or `x`

You can also specify more [granular semver ranges](#).

---

Last modified December 29, 2015

Found a typo? Send a [pull request!](#)

## Working with scoped packages

Scopes are like namespaces for npm modules. If a package's name begins with `@`, then it is a scoped package. The scope is everything in between the `@` and the slash.

```
@scope/project-name
```

Each npm user has their own scope.

```
@username/project-name
```

You can find more in depth information about scopes in the [CLI documentation](#).

## Update npm and log in

You need a version of npm greater than `2.7.0`, and you'll need to log in to npm again on the command line if this is your first time using scoped modules.

```
sudo npm install -g npm
npm login
```

## Initializing a scoped package

To create a scoped package, you simply use a package name that starts with your scope.

```
{
  "name": "@username/project-name"
}
```

If you use `npm init`, you can add your scope as an option to that command.

```
npm init --scope=username
```

If you use the same scope all the time, you will probably want to set this option in your [.npmrc](#) file.

```
npm config set scope username
```

## Publishing a scoped package

Scoped packages are private by default. To publish private modules, you need to be a paid [private modules](#) user.

However, public scoped modules are free and don't require a paid subscription. To publish a public scoped module, set the access option when publishing it. This option will remain set for all subsequent publishes.

```
npm publish --access=public
```

## Using a scoped package

To use a scoped package, you simply include the scope wherever you use the package name.

In `package.json`:

```
{
  "dependencies": {
    "@username/project-name": "^1.0.0"
  }
}
```

```
}  
}
```

On the command line:

```
npm install @username/project-name --save
```

In a `require` statement:

```
var projectName = require("@username/project-name")
```

For information about using scoped private modules, visit [npmjs.com/private-modules](https://npmjs.com/private-modules).

---

Last modified June 17, 2016

Found a typo? Send a [pull request!](#)

## Using dist-tags

Tags are a supplement to [semver](#) (e.g., v0.12) for organizing and labeling different versions of packages. In addition to being more human-readable, tags allow publishers to distribute their packages more effectively.

## Adding tags

To add a tag to a specific version of your package, use `npm dist-tag add <pkg>@<version> [<tag>]`. See [the CLI docs](#) for more information.

## Publishing with tags

By default, `npm publish` will tag your package with the `latest` tag. If you use the `--tag` flag, you can specify another tag to use. For example, the following will publish your package with the `beta` tag:

```
npm publish --tag beta
```

## Installing with tags

Like `npm publish`, `npm install <pkg>` will use the `latest` tag by default. To override this behavior, use `npm install <pkg>@<tag>`. The following example will install the `somepkg` at the version that has been tagged with `beta`.

```
npm install somepkg@beta
```

## Caveats

Because dist-tags share the same namespace with semver, avoid using any tag names that may cause a conflict. The best practice is to avoid using tags beginning with a number or the letter "v".

---

Last modified January 19, 2016

Found a typo? Send a [pull request!](#)

## Packages and Modules

One of the key steps in becoming immersed in an ecosystem is learning its vocabulary. Node.js and npm have very specific definitions of packages and modules, which are easy to mix up. We'll discuss those definitions here, make them distinct, and explain why certain default files are named the way they are.

## Quick Summary

- A **package** is a file or directory that is described by a `package.json` . This can happen in a bunch of different ways! For more info, see ["What is a package ?"](#), below.
- A **module** is any file or directory that can be loaded by Node.js' `require()` . Again, there are several configurations that allow this to happen. For more info, see ["What is a module ?"](#), below.

## What is a package ?

A package is any of the following:

- a) a folder containing a program described by a `package.json` file
- b) a gzipped tarball containing (a)
- c) a url that resolves to (b)
- d) a `<name>@<version>` that is published on the registry with (c)
- e) a `<name>@<tag>` that points to (d)
- f) a `<name>` that has a `latest` tag satisfying (e)
- g) a `git` url that, when cloned, results in (a).

Noting all these **package** possibilities, it follows that even if you never publish your package to the public registry, you can still get a lot of benefits of using npm:

- if you just want to write a node program, and/or
- if you also want to be able to easily install it elsewhere after packing it up into a tarball

Git urls can be of the form:

```
git://github.com/user/project.git#commit-ish
git+ssh://user@hostname:project.git#commit-ish
git+http://user@hostname/project/blah.git#commit-ish
git+https://user@hostname/project/blah.git#commit-ish
```

The `commit-ish` can be any tag, sha, or branch which can be supplied as an argument to `git checkout` . The default is `master` .

## What is a module ?

A module is anything that can be loaded with `require()` in a Node.js program. The following are all examples of things that can be loaded as modules:

- A folder with a `package.json` file containing a `main` field.
- A folder with an `index.js` file in it.
- A JavaScript file.

### Most npm packages are modules

Generally, npm packages that are used in Node.js program are loaded with `require` , making them modules. However, there's no requirement that an npm package be a module!

Some packages, e.g., `cli` packages, only contain an executable command-line interface and don't provide a `main` field for use in Node.js programs. These packages are *not* modules.

Almost all npm packages (at least, those that are Node programs) *contain* many modules within them (because every file they load with `require()` is a module).

In the context of a Node program, the **module** is also the thing that was loaded *from* a file. For example, in the following program:

```
var req = require('request')
```

we might say that "The variable `req` refers to the `request` module".

## File and Directory Names in the Node.js and npm Ecosystem

- So, why is it the `node_modules` folder, but `package.json` file?
- Why not `node_packages` or `module.json` ?

The `package.json` file defines the package. (See ["What is a package ?"](#), above.)

The `node_modules` folder is the place Node.js looks for modules. (See ["What is a module ?"](#), above.)

For example, if you create a file at `node_modules/foo.js` and then had a program that did `var f = require('foo.js')`, it would load the module. However, `foo.js` is not a "package" in this case because it does not have a `package.json`.

Alternatively, if you create a package which does not have an `index.js` or a `"main"` field in the `package.json` file, then it is not a module. Even if it's installed in `node_modules`, it can't be an argument to `require()`.

---

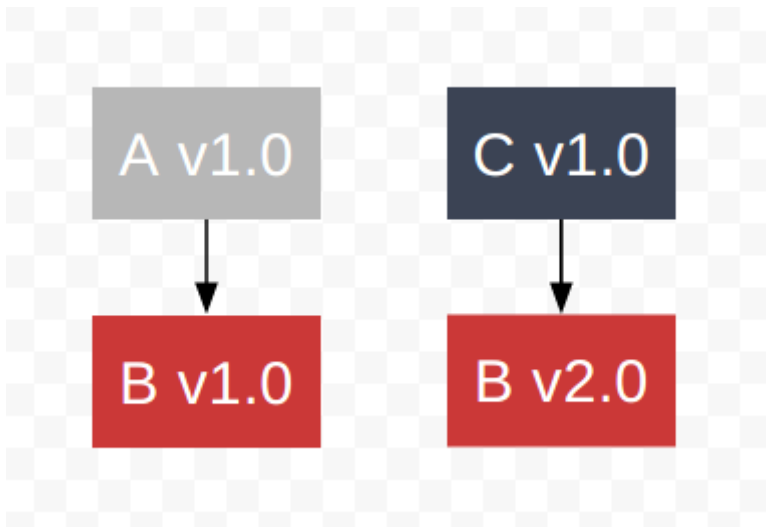
Last modified December 02, 2016

Found a typo? Send a [pull request!](#)

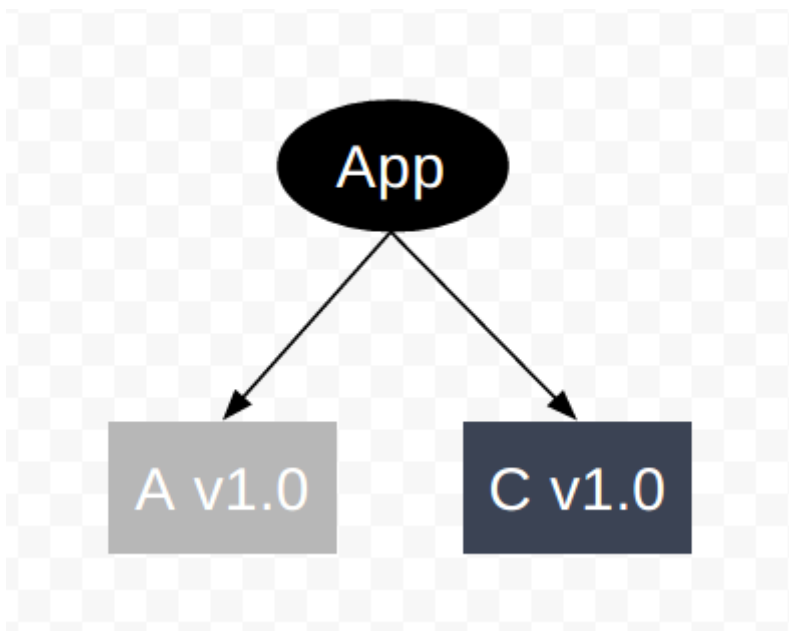
## npm v2 Dependency Resolution

### Example - [Explore on Github](#)

Imagine there are three modules: A, B, and C. A requires B at v1.0, and C also requires B, but at v2.0. We can visualize this like so:

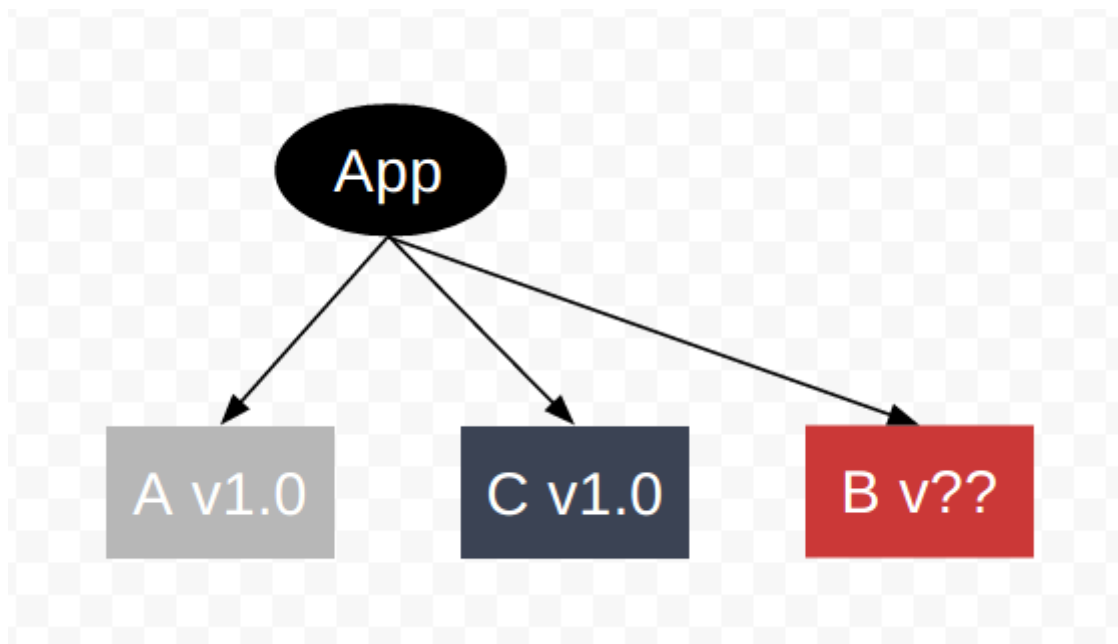


Now, let's create an application that requires both module A and module C.

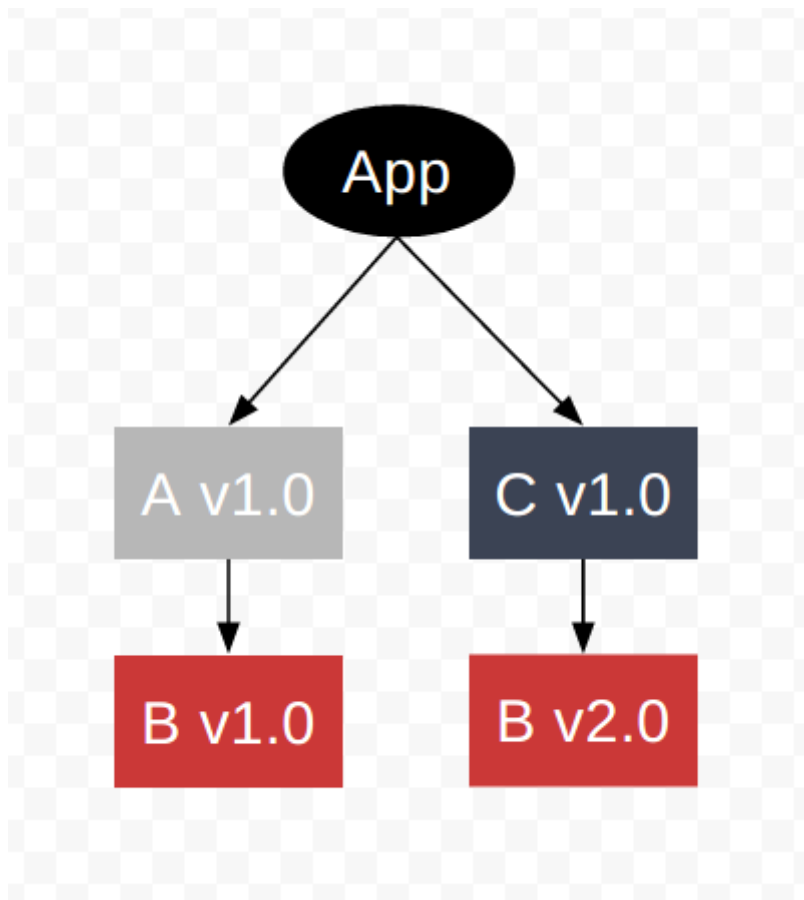


# Dependency Hell

A package manager would need to provide a version of module B. In all other runtimes prior to Node.js, this is what a package manager would try to do. This is dependency hell:



Instead of attempting to resolve module B to a single version, npm puts both versions of module B into the tree, each version nested under the module that requires it.



In the terminal, this looks like this:



```
(jessie)ag_dubs@localhost:~/Projects/npm-sandbox/npm2/example1$ npm install mod-a --save
npm WARN package.json example1@1.0.0 No repository field.
mod-a@1.0.0 node_modules/mod-a
└─ mod-b@1.0.0
(jessie)ag_dubs@localhost:~/Projects/npm-sandbox/npm2/example1$ npm install mod-c --save
npm WARN package.json example1@1.0.0 No repository field.
mod-c@1.0.0 node_modules/mod-c
└─ mod-b@2.0.0
(jessie)ag_dubs@localhost:~/Projects/npm-sandbox/npm2/example1$ tree -d node_modules/
node_modules/
├── mod-a
│   └── node_modules
│       └── mod-b
├── mod-c
│   └── node_modules
│       └── mod-b
```

You can list the dependencies and still see their relationships using `npm ls` :

```
(jessie)ag_dubs@localhost:~/Projects/npm-sandbox/npm2/example1$ npm ls
example1@1.0.0 /home/ag_dubs/Projects/npm-sandbox/npm2/example1
├── mod-a@1.0.0
│   └── mod-b@1.0.0
├── mod-c@1.0.0
│   └── mod-b@2.0.0
```

If you want to just see your primary dependencies, you can use:

```
npm ls --depth=0
```

```
(jessie)ag_dubs@localhost:~/Projects/npm-sandbox/npm2/example1$ npm ls --depth=0
example1@1.0.0 /home/ag_dubs/Projects/npm-sandbox/npm2/example1
├── mod-a@1.0.0
└── mod-c@1.0.0
```

## npm and the Node.js Module Loader

However, npm doing this is *not enough*. Despite the fact that their nested locations allow for the coexistence of two versions of the same module, most module loaders are unable to load two different versions of the same module into memory. Luckily, the Node.js module loader is written for exactly this situation, and can easily load both versions of the module in a way that they do not conflict with each other.

How is it that npm and the node module loader are so wonderfully symbiotic? They were both written in large part by the same person, npm, Inc. CEO, Isaac Z. Schlueter. Like 2 sides of the same piece of paper, npm and the Node.js module loader are what make Node.js a uniquely well-suited runtime for dependency management.

---

Last modified December 29, 2015

Found a typo? Send a [pull request!](#)

## npm v3 Dependency Resolution

npm3 resolves dependencies differently than npm2.

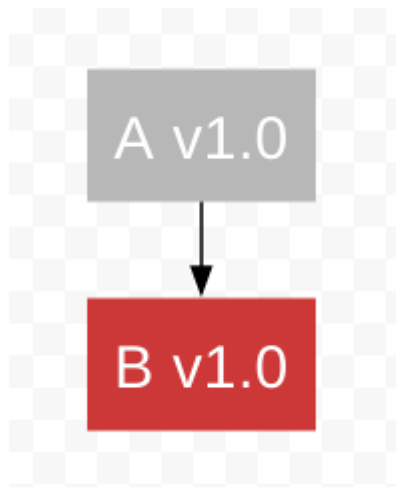
While npm2 installs all dependencies in a nested way, npm3 tries to mitigate the deep trees and redundancy that such nesting causes. npm3 attempts this by installing some secondary dependencies (dependencies of dependencies) in a flat way, in the same directory as the primary dependency that requires it.

The key major differences are:

- position in the directory structure no longer predicts the type (primary, secondary, etc) a dependency is
- dependency resolution depends on *install order*, or the order in which things are installed will change the `node_modules` directory tree structure

## Example - [Explore on Github](#)

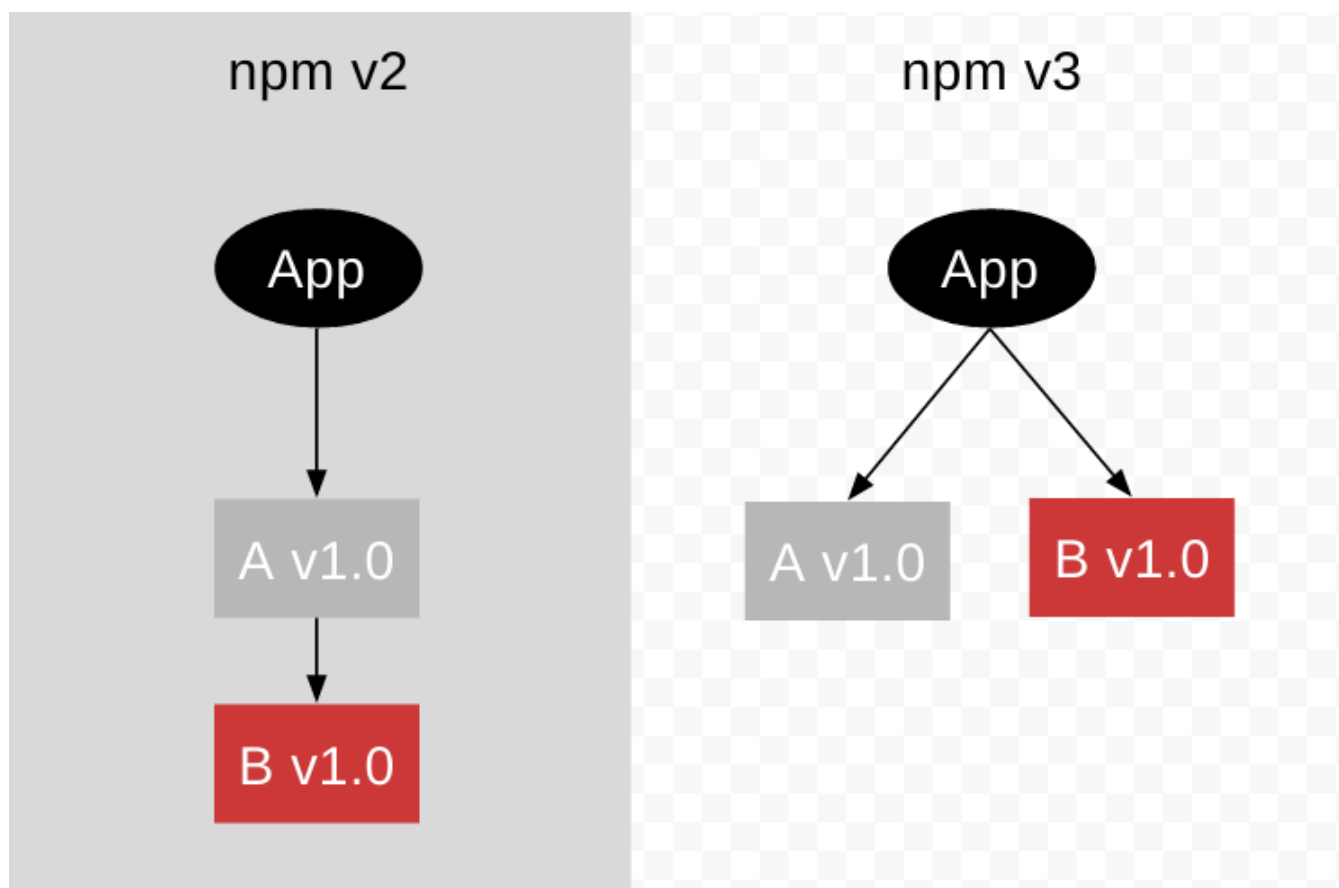
Imagine we have a module, A. A requires B.



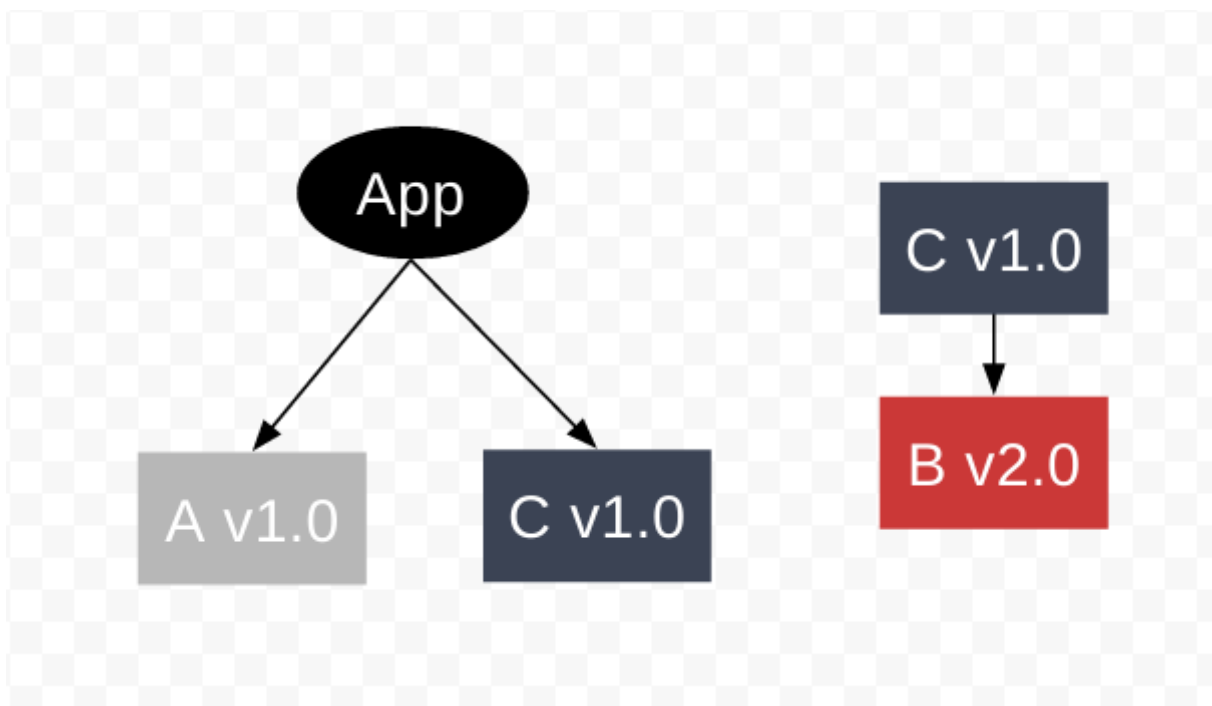
Now, let's create an application that requires module A.

On `npm install`, npm v3 will install both module A and its dependency, module B, inside the `/node_modules` directory, flat.

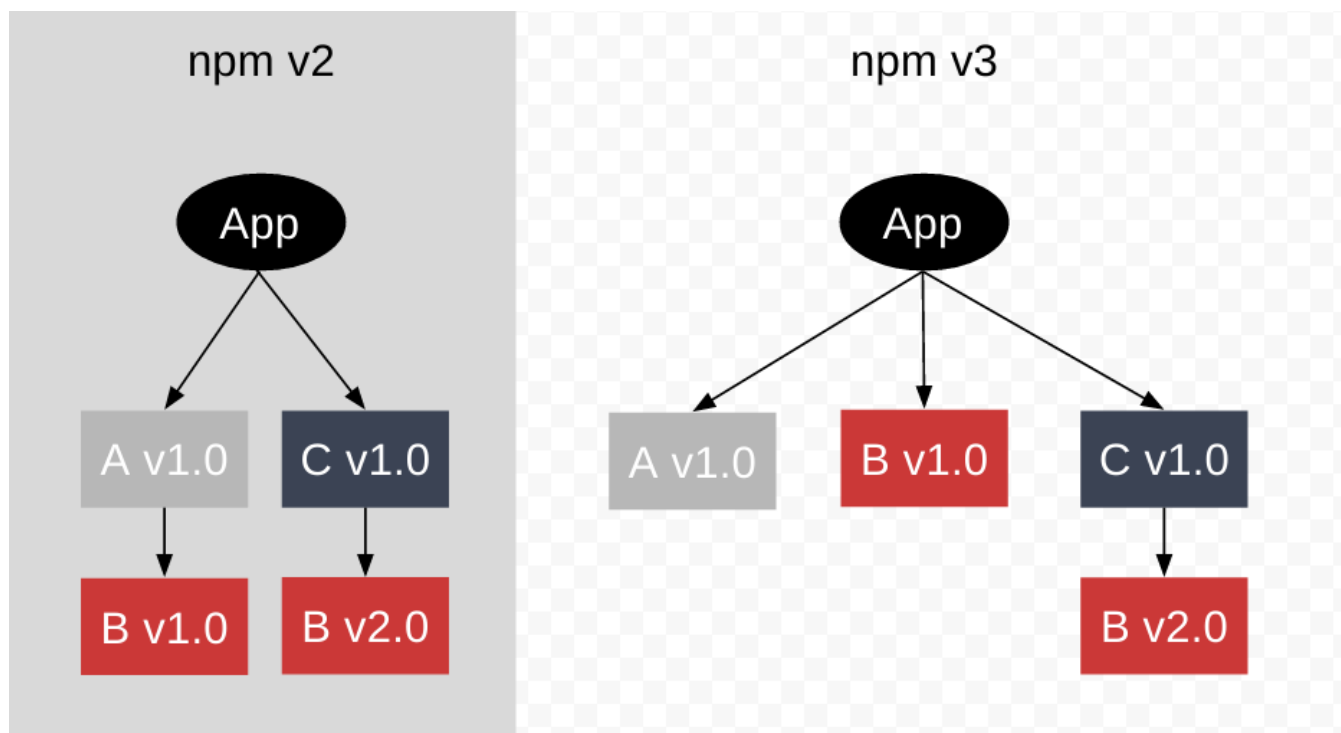
In npm v2 this would have happened in a nested way.



Now, let's say we want to require another module, C. C requires B, but at another version than A.



However, since B v1.0 is already a top-level dep, we cannot install B v2.0 as a top level dependency. npm v3 handles this by defaulting to npm v2 behavior and nesting the new, different, module B version dependency under the module that requires it -- in this case, module C.



In the terminal, this looks like this:

```
(jessie)ag_dubs@localhost:~/Projects/npm-sandbox/npm3/example1$ npm install mod-a --save
example1@1.0.0 /home/ag_dubs/Projects/npm-sandbox/npm3/example1
└─┬─ mod-a@1.0.0
  └─ mod-b@1.0.0

npm WARN example1@1.0.0 No description
npm WARN example1@1.0.0 No repository field.
(jessie)ag_dubs@localhost:~/Projects/npm-sandbox/npm3/example1$ npm install mod-c --save
example1@1.0.0 /home/ag_dubs/Projects/npm-sandbox/npm3/example1
└─┬─ mod-c@1.0.0
  └─ mod-b@2.0.0

npm WARN example1@1.0.0 No description
npm WARN example1@1.0.0 No repository field.
(jessie)ag_dubs@localhost:~/Projects/npm-sandbox/npm3/example1$ tree -d node_modules/
node_modules/
├── mod-a
├── mod-b
├── mod-c
│   └── node_modules
│       └── mod-b
5 directories
(jessie)ag_dubs@localhost:~/Projects/npm-sandbox/npm3/example1$ npm ls
example1@1.0.0 /home/ag_dubs/Projects/npm-sandbox/npm3/example1
└─┬─ mod-a@1.0.0
  └─┬─ mod-b@1.0.0
    └─ mod-c@1.0.0
      └─ mod-b@2.0.0
```

You can list the dependencies and still see their relationships using `npm ls` :

```
(jessie)ag_dubs@localhost:~/Projects/npm-sandbox/npm3/example1$ npm ls
example1@1.0.0 /home/ag_dubs/Projects/npm-sandbox/npm3/example1
└─┬─ mod-a@1.0.0
  └─┬─ mod-b@1.0.0
    └─ mod-c@1.0.0
      └─ mod-b@2.0.0
```

If you want to just see your primary dependencies, you can use:

```
npm ls --depth=0
```

```
(jessie)ag_dubs@localhost:~/Projects/npm-sandbox/npm3/example1$ npm ls --depth=0
example1@1.0.0 /home/ag_dubs/Projects/npm-sandbox/npm3/example1
├─ mod-a@1.0.0
└─ mod-c@1.0.0
```

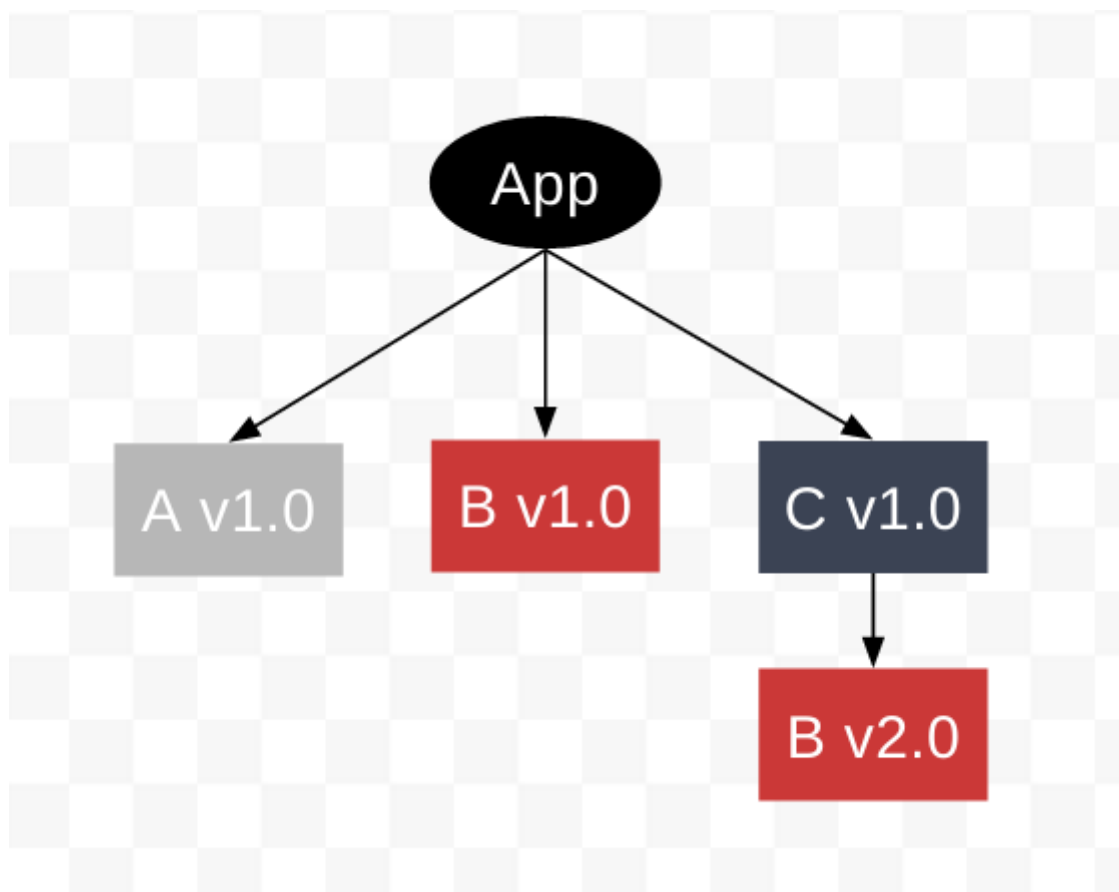
Last modified December 30, 2015

Found a typo? Send a [pull request!](#)

## npm3 Duplication and Deduplication

Let's continue with our example before. Currently we have an application that depends on 2 modules:

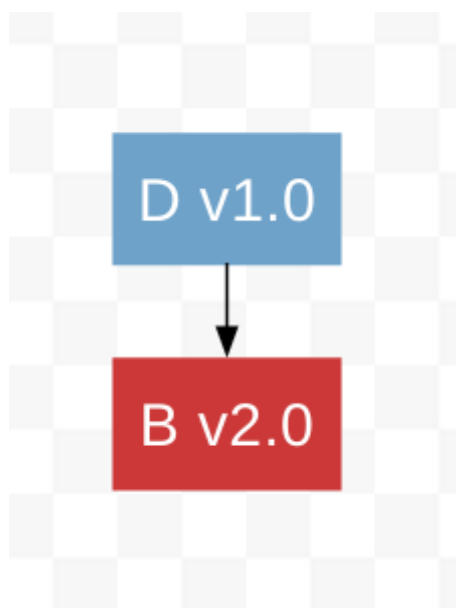
- Module-A, depends on Module B v1.0
- Module-C, depends on Module B v2.0



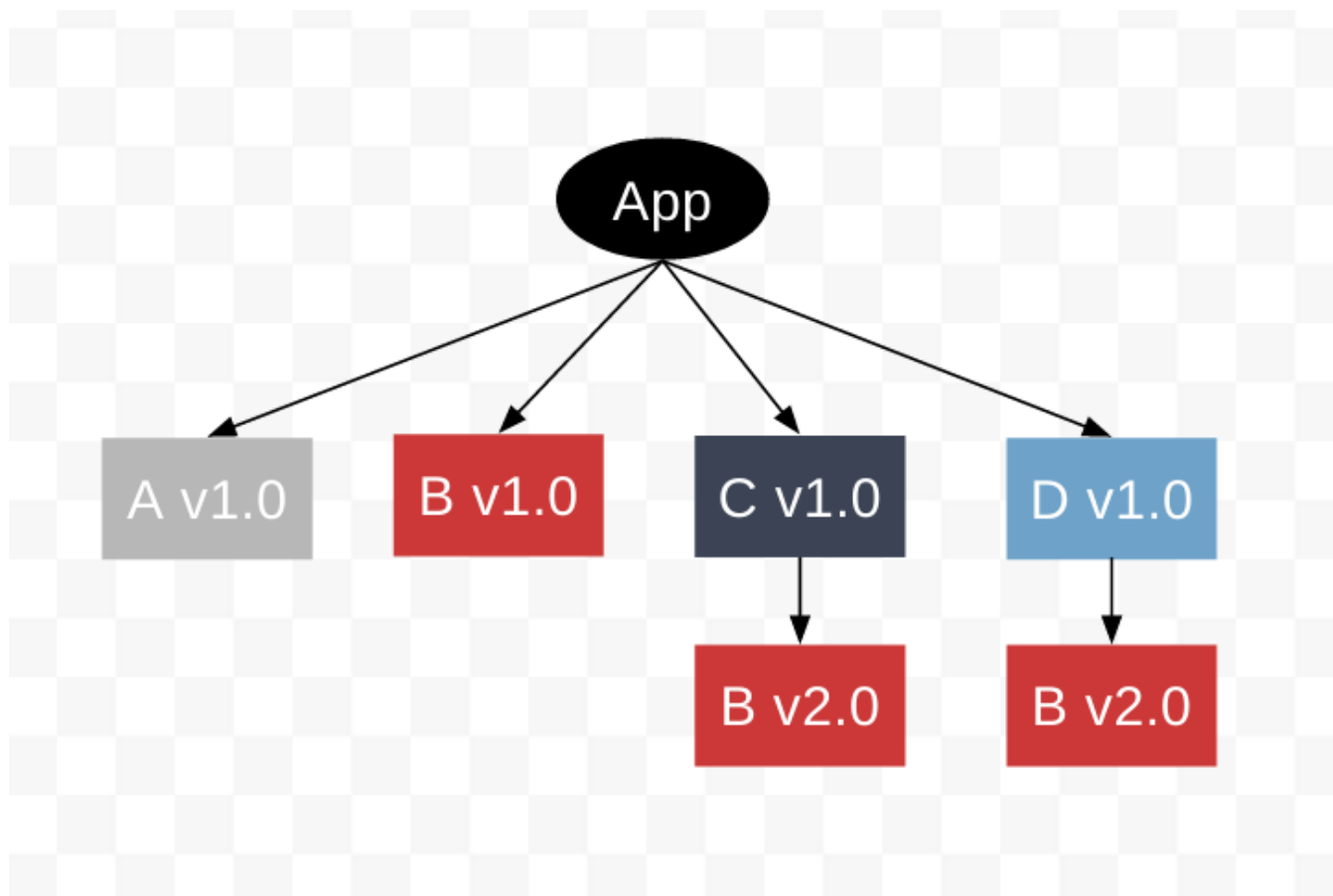
Now we ask ourselves, what happens if we install another module that depends on Module B v1.0? or Module B v2.0?

## Example

Ok, so let's say we want to depend on another package, module D. Module D depends on Module B v2.0, just like Module C.



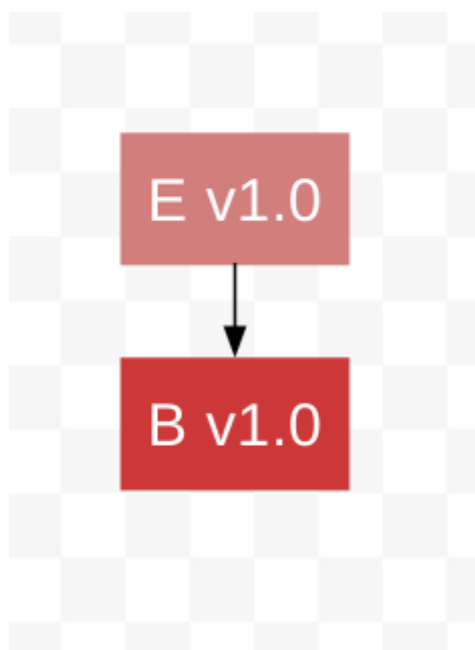
Because B v1.0 is already a top-level dependency, we cannot install B v2.0 as a top level dependency. Therefore Module B v2.0 is installed as a nested dependency of Module D, even though we already have a copy installed, nested beneath Module C.



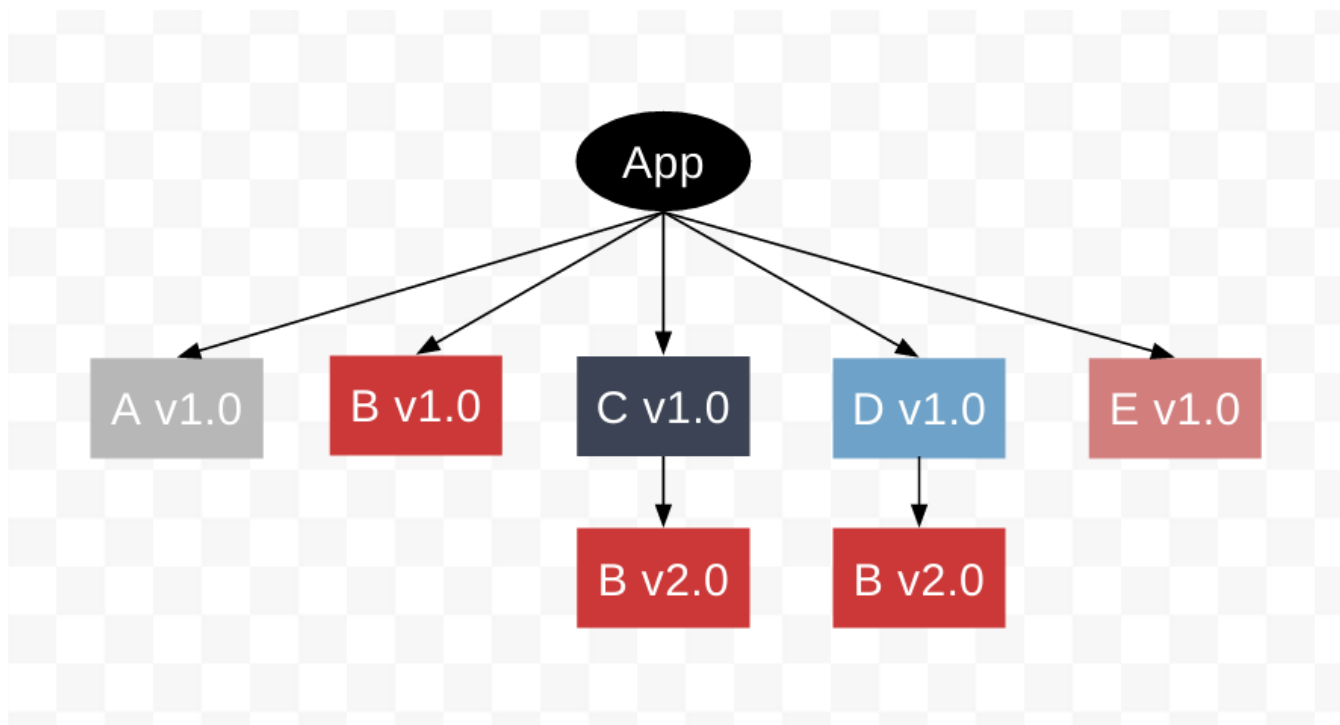
If a secondary dependency is required by 2+ modules, but was not installed as a top-level dependency in the directory hierarchy, it will be duplicated and nested beneath the primary dependency.

However, if a secondary dependency is required by 2+ modules, but *is* installed as a top-level dependency in the directory hierarchy, it will *not* be duplicated, and will be shared by the primary dependencies that require it.

For example, let's say we now want to depend on Module E. Module E, like Module A, depends on Module B v1.0.



Because B v1.0 is already a top-level dependency, we do not need to duplicate and nest it. We simply install Module E and it shares Module B v1.0 with Module A.



This appears like this in the terminal:

```

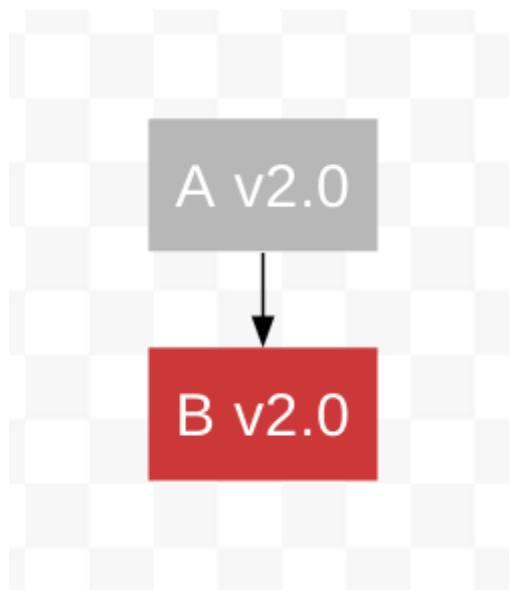
npm WARN example3@1.0.0 No repository field.
(jessie)ag_dubs@localhost:~/Projects/npm-sandbox/npm3/example3$ tree -d node_modules/
node_modules/
├── mod-a
├── mod-b
├── mod-c
│   └── node_modules
│       └── mod-b
├── mod-d
│   └── node_modules
│       └── mod-b
└── mod-e

8 directories
(jessie)ag_dubs@localhost:~/Projects/npm-sandbox/npm3/example3$ npm install mod-e --save
example3@1.0.0 /home/ag_dubs/Projects/npm-sandbox/npm3/example3
└── mod-e@1.0.0

npm WARN example3@1.0.0 No description
npm WARN example3@1.0.0 No repository field.
(jessie)ag_dubs@localhost:~/Projects/npm-sandbox/npm3/example3$ tree -d node_modules/
node_modules/
├── mod-a
├── mod-b
├── mod-c
│   └── node_modules
│       └── mod-b
├── mod-d
│   └── node_modules
│       └── mod-b
└── mod-e

9 directories
  
```

Now-- what happens if we update Module A to v2.0, which depends on Module B v2.0, *not* Module B v1.0?

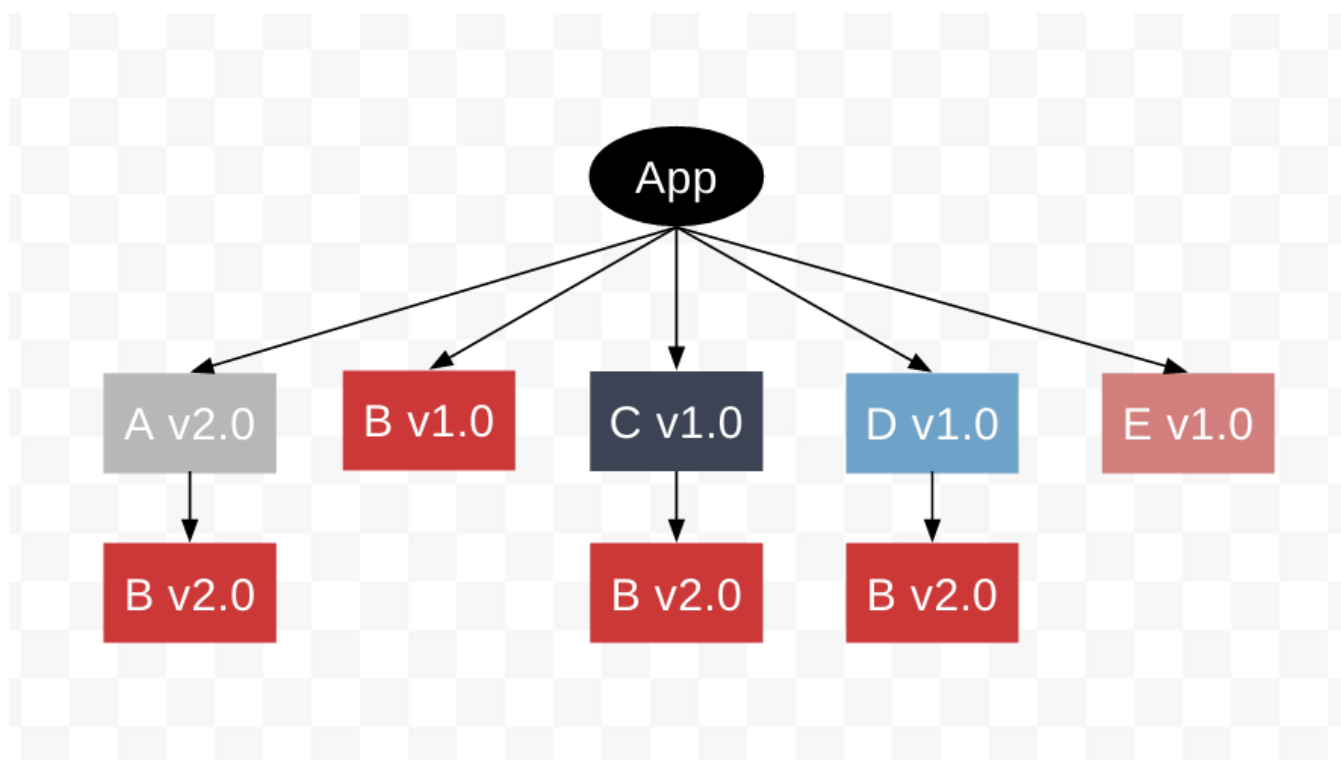


The key is to remember that install order matters.

Even though Module A was installed first (as v1.0) via our `package.json` (because it is ordered alphabetically), using the interactive `npm install` command means that Module A v2.0 is the last package installed.

As a result, npm3 does the following things when we run `npm install mod-a@2 --save` :

- it removes Module A v1.0
- it installs Modules A v2.0
- it leaves Module Bv1.0 because Module E v1.0 still depends on it
- it installs Module Bv2.0 as a nested dependency under Module A v2.0, since Module B v1.0 is already occupying the top level in the directory hierarchy



This looks like this in the terminal:

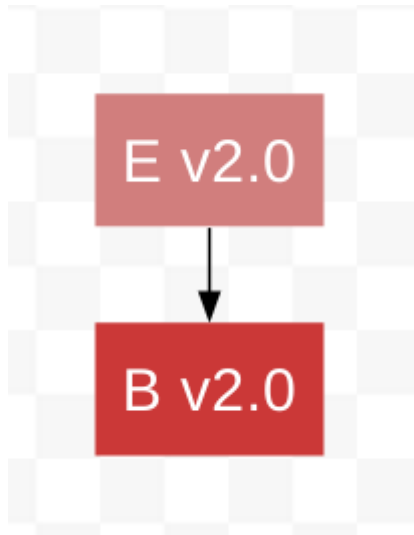


```
(jessie)ag_dubs@localhost:~/Projects/npm-sandbox/npm3/example4$ npm install mod-a@2 --save
example3@1.0.0 /home/ag_dubs/Projects/npm-sandbox/npm3/example4
├── mod-a@2.0.0
│   └── mod-b@2.0.0
├── mod-c@1.0.0
│   └── mod-b@2.0.0
├── mod-d@1.0.0
│   └── mod-b@2.0.0
└── mod-b@2.0.0

npm WARN example3@1.0.0 No description
npm WARN example3@1.0.0 No repository field.
(jessie)ag_dubs@localhost:~/Projects/npm-sandbox/npm3/example4$ tree -d node_modules/
node_modules/
├── mod-a
│   └── node_modules
│       └── mod-b
├── mod-b
├── mod-c
│   └── node_modules
│       └── mod-b
├── mod-d
│   └── node_modules
│       └── mod-b
└── mod-e

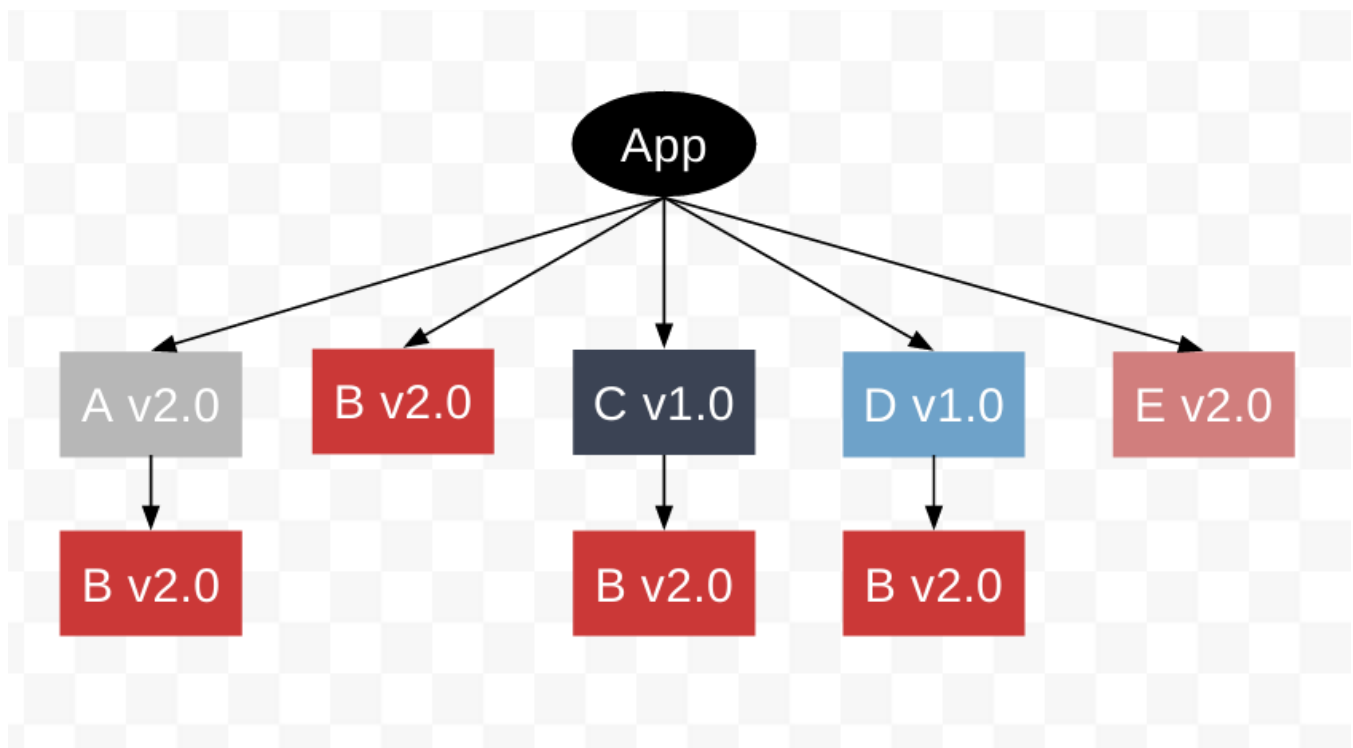
11 directories
```

Finally, let's also update Module E to v2.0, which also depends on Module B v2.0 instead of Module B v1.0, just like the Module A update.



npm3 performs the following things:

- it removes Module E v1.0
- it installs Module E v2.0
- it removes Module B v1.0 because nothing depends on it anymore
- it installs Module B v2.0 in the top level of the directory because there is no other version of Module B there



This looks like this in the terminal:

```

(jessie)ag_dubs@localhost:~/Projects/npm-sandbox/npm3/example5$ npm install mod-e@2 --save
example3@1.0.0 /home/ag_dubs/Projects/npm-sandbox/npm3/example5
├── mod-a@2.0.0
│   └── mod-b@2.0.0
├── mod-c@1.0.0
│   └── mod-b@2.0.0
├── mod-d@1.0.0
│   └── mod-b@2.0.0
├── mod-e@2.0.0
└── mod-b@2.0.0

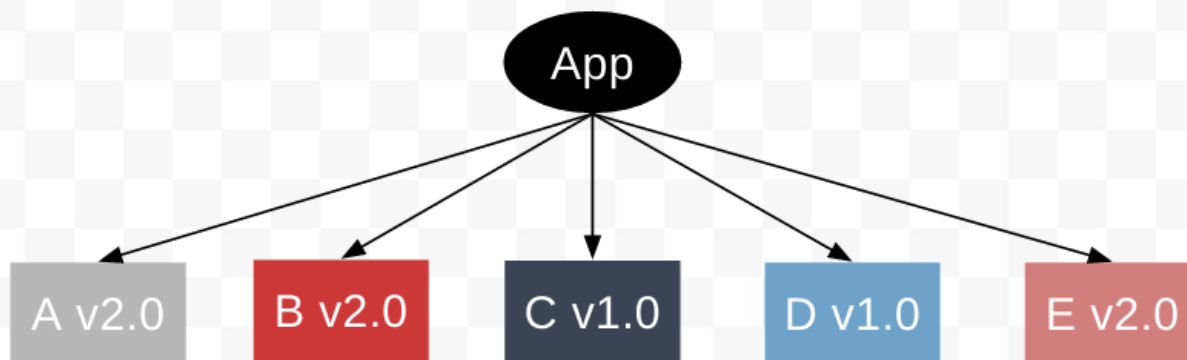
npm WARN example3@1.0.0 No description
npm WARN example3@1.0.0 No repository field.
(jessie)ag_dubs@localhost:~/Projects/npm-sandbox/npm3/example5$ tree -d node_modules/
node_modules/
├── mod-a
│   └── node_modules
│       └── mod-b
├── mod-b
├── mod-c
│   └── node_modules
│       └── mod-b
├── mod-d
│   └── node_modules
│       └── mod-b
└── mod-e

11 directories
  
```

Now, this is clearly not ideal. We have Module B v2.0 in nearly every directory. To get rid of duplication, we can run:

```
npm dedupe
```

This command resolves all of the packages dependencies on Module B v2.0 by redirecting them to the top level copy of Module B v2.0 and removes all the nested copies.



This looks like this in the terminal:

```
(jessie)ag_dubs@localhost:~/Projects/npm-sandbox/npm3/example5$ npm dedupe
- mod-b@2.0.0 node_modules/mod-a/node_modules/mod-b
- mod-b@2.0.0 node_modules/mod-c/node_modules/mod-b
- mod-b@2.0.0 node_modules/mod-d/node_modules/mod-b
(jessie)ag_dubs@localhost:~/Projects/npm-sandbox/npm3/example5$ tree -d node_modules/
node_modules/
├── mod-a
├── mod-b
├── mod-c
├── mod-d
└── mod-e

5 directories
```

Last modified December 30, 2015

Found a typo? Send a [pull request!](#)

## npm3 Non-determinism

As stated a few pages back in our example:

your ``node_modules`` directory structure and  
therefore your **dependency tree**, are dependent on

# INSTALL ORDER

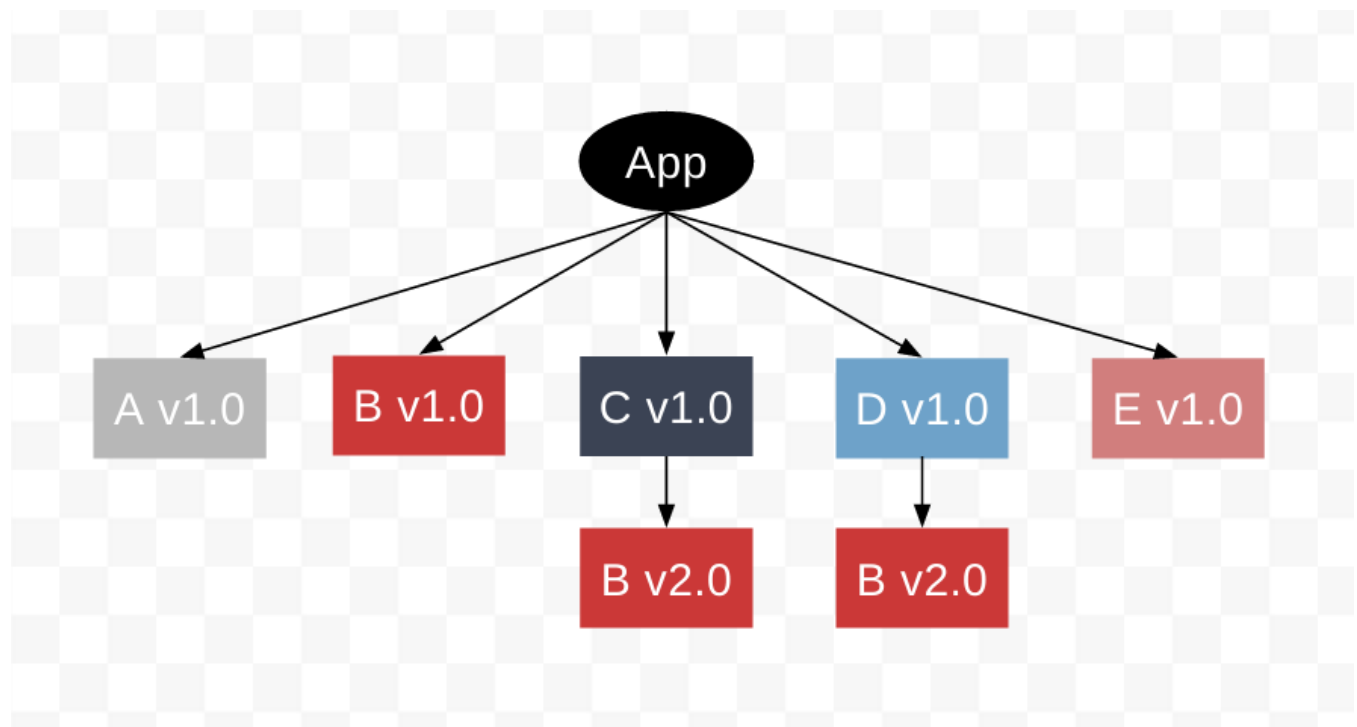
If you, and your development team, use a `package.json`, as well as the interactive `npm install` command to add pkgs (like most teams using npm do), it is likely that you will run into a situation where your local `node_modules` directory will differ from both your coworkers' `node_modules` directories, as well as the `node_modules` directories on your staging, testing, or production servers.

In short? **npm3 does not install dependencies in a deterministic way.**

That's probably not a comforting statement to read, but in this article we'll discuss why this happens, as well as assure you that it has no implications for your application, as well as explain the steps to reliably (re)create a single, consistent, `node_modules` directory, should you want to do that.

## Example

Let's jump back to an example application from a few examples ago:



In this example, our app has the following `package.json` :

```
{
  "name": "example3",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "dependencies": {
    "mod-a": "^1.0.0",
    "mod-c": "^1.0.0",
    "mod-d": "^1.0.0",
    "mod-e": "^1.0.0"
  }
}
```

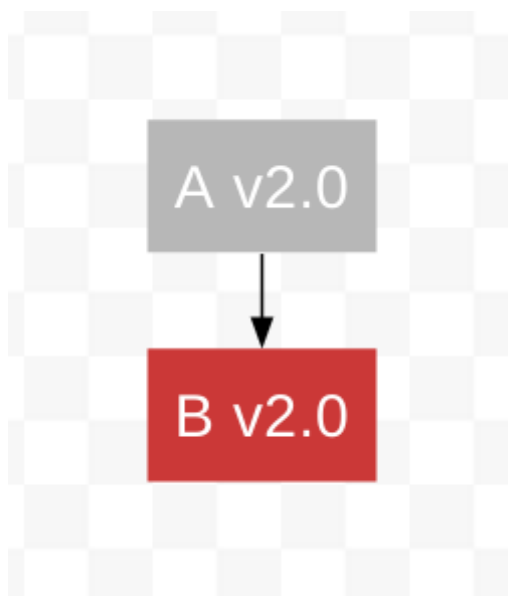
On an `npm install` we will see this in our terminal:

```
(jessie)ag_dubs@localhost:~/Projects/npm-sandbox/npm3/example3$ npm install
example3@1.0.0 /home/ag_dubs/Projects/npm-sandbox/npm3/example3
├── mod-a@1.0.0
│   └── mod-b@1.0.0
│       ├── mod-c@1.0.0
│       │   └── mod-b@2.0.0
│       └── mod-d@1.0.0
│           └── mod-b@2.0.0
└── mod-e@1.0.0

npm WARN example3@1.0.0 No description
npm WARN example3@1.0.0 No repository field.
(jessie)ag_dubs@localhost:~/Projects/npm-sandbox/npm3/example3$ tree -d node_modules/
node_modules/
├── mod-a
├── mod-b
├── mod-c
│   └── node_modules
│       └── mod-b
├── mod-d
│   └── node_modules
│       └── mod-b
└── mod-e

9 directories
```

Now, let's say a developer on our team decides to complete a feature that requires that they update Module A to v2.0, which now has a dependency on Module B v2.0, instead of, as previously, Module B v1.0.



Our developer uses the interactive `npm install` command to install the new version of Module A, and save it to the `package.json` :

```
npm install mod-a@2 --save
```

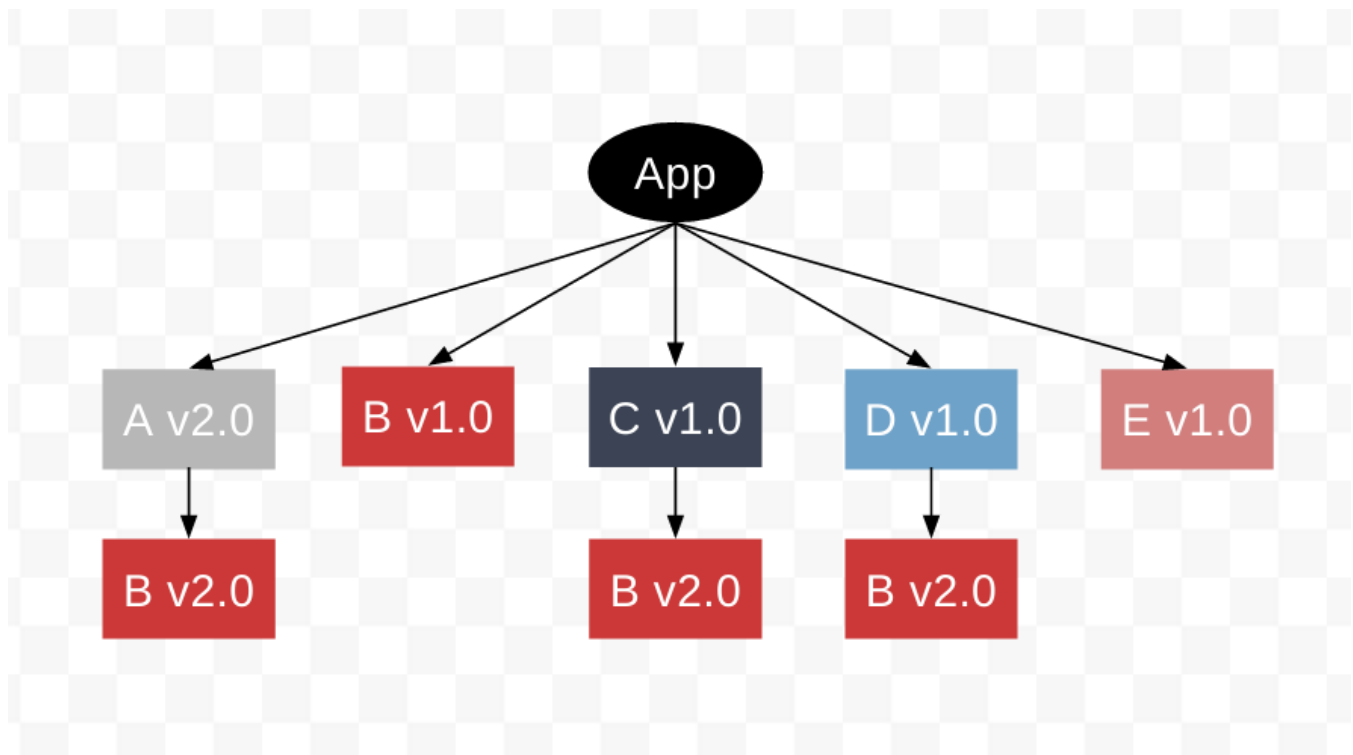
The terminal outputs this:

```
(jessie)ag_dubs@localhost:~/Projects/npm-sandbox/npm3/example3$ npm install mod-a@2 --save
example3@1.0.0 /home/ag_dubs/Projects/npm-sandbox/npm3/example3
├── mod-a@2.0.0
│   └── mod-b@2.0.0
├── mod-c@1.0.0
│   └── mod-b@2.0.0
├── mod-d@1.0.0
│   └── mod-b@2.0.0
└── mod-b@2.0.0

npm WARN example3@1.0.0 No description
npm WARN example3@1.0.0 No repository field.
(jessie)ag_dubs@localhost:~/Projects/npm-sandbox/npm3/example3$ tree -d node_modules/
node_modules/
├── mod-a
│   └── node_modules
│       └── mod-b
├── mod-b
├── mod-c
│   └── node_modules
│       └── mod-b
├── mod-d
│   └── node_modules
│       └── mod-b
└── mod-e

11 directories
```

We now have something that looks like this:



Now let's say that our developer finished the feature requiring the new version of Module A and pushes the application to a testing server that runs `npm install` on the new `package.json` :

```
{
  "name": "example3",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
```

```

"dependencies": {
  "mod-a": "^2.0.0",
  "mod-c": "^1.0.0",
  "mod-d": "^1.0.0",
  "mod-e": "^1.0.0"
}

```

The testing server's log shows this:

```

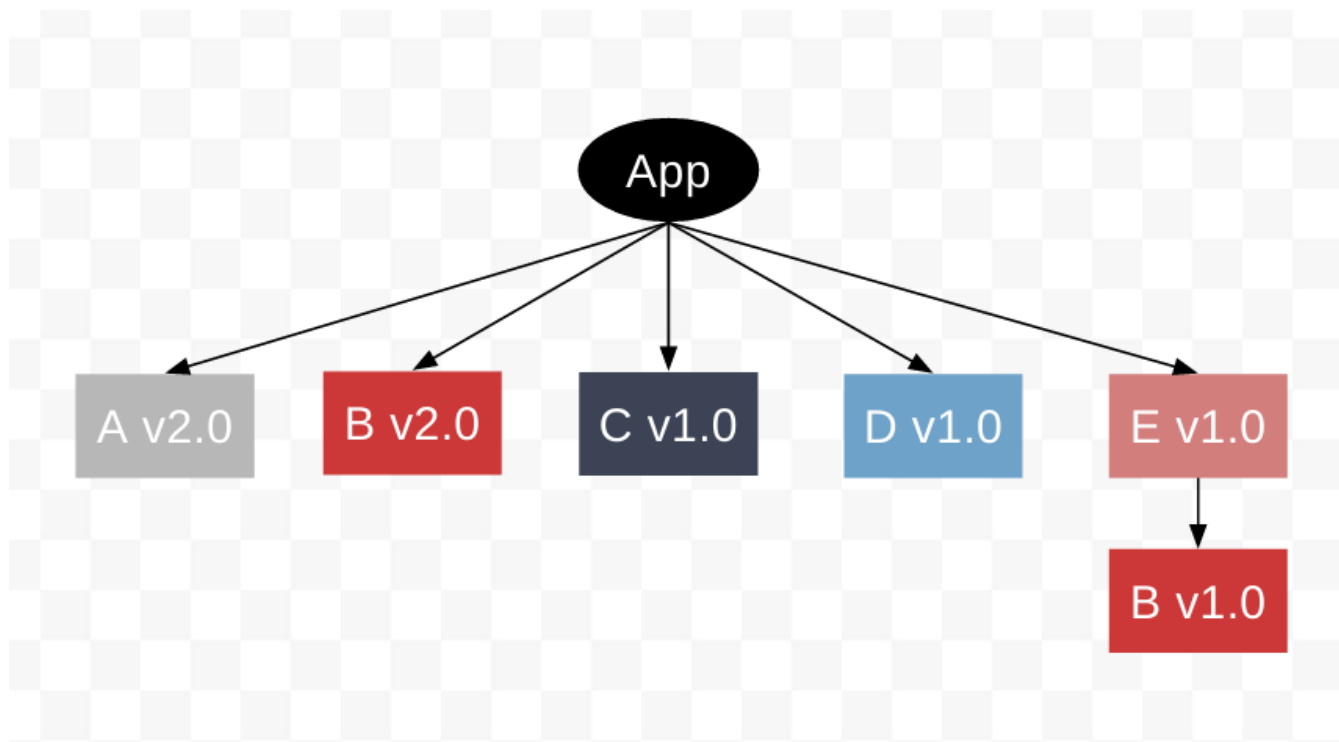
(jessie)ag_dubs@localhost:~/Projects/npm-sandbox/npm3/example3$ npm install
example3@1.0.0 /home/ag_dubs/Projects/npm-sandbox/npm3/example3
├── mod-a@2.0.0
├── mod-b@2.0.0
├── mod-c@1.0.0
├── mod-d@1.0.0
├── mod-e@1.0.0
└── mod-b@1.0.0

npm WARN example3@1.0.0 No description
npm WARN example3@1.0.0 No repository field.
(jessie)ag_dubs@localhost:~/Projects/npm-sandbox/npm3/example3$ tree -d node_modules/
node_modules/
├── mod-a
├── mod-b
├── mod-c
├── mod-d
├── mod-e
└── node_modules
    └── mod-b

7 directories

```

Which, when visualized, looks like this:



Whoa, what?! This tree is completely different than the tree that exists on our developer's local machine. What happened?

**Remember: install order matters.**

When our developer updated Module A using the interactive `npm install` Module A v2.0 was functionally the **last** package installed. Because our developer had done an `npm install` when they first started working on the project, all modules listed in the `package.json` were already installed in the `node_modules` folder. Then Module A v2.0 was installed.

It follows, then, that Module Bv1.0, a top level dependency because of Module A v1.0, then anchored by Module E v1.0, remains a top level dependency. Because Module Bv1.0 occupies the top-level, no other version of Module B can-- therefore, Module Bv2.0 remains a nested dependency under Module C v1.0 and Module D v1.0, and becomes a nested dependency for the new Module A v2.0 dependency.

Let's consider what happened on the testing server. The project was pulled into a fresh directory, i.e. does not have a pre-existing `node_modules` directory. Then `npm install` is run, perhaps by a deploy script, to install dependencies from the `package.json` .

This `package.json` now has Module A v2.0 listed in it, and thanks to alphabetical order (enforced by the `npm install` command), is now installed **first**, instead of **last**.

When Module A v2.0 is installed first, in a clear `node_modules` directory, its dependencies are the **first candidates** for the top-level position. As a result, Module B v2.0 is installed in the top-level of the `node_modules` directory.

Now, when it is time to install Module E v1.0, its dependency, Module B v1.0, cannot occupy the top-level of the `node_modules` directory, because Module B v2.0 is already there. Therefore, it is nested under Module E v1.0.

## Do different dependency tree structures affect my app?

No! Even though the trees are different, both sufficiently install and point all your dependencies at all their dependencies, and so on, down the tree. You still have everything you need, it just happens to be in a different configuration.

## I want my `node_modules` directory to be the same. How can I do that?

The `npm install` command, when used exclusively to install packages from a `package.json` , will **always produce the same tree**. This is because install order from a `package.json` is **always** alphabetical. Same install order means that you will get the same tree.

You can reliably get the same dependency tree by removing your `node_modules` directory and running `npm install` whenever you make a change to your `package.json` .

---

Last modified December 30, 2015

Found a typo? Send a [pull request!](#)

## Working with private modules

With npm private modules, you can use the npm registry to host your own private code and the npm command line to manage it. This makes it easy to use public modules like Express and Browserify side-by-side with your own private code.

## Before we start

You need a version of npm greater than `2.7.0` , and you'll need to log in to npm again.

```
sudo npm install -g npm
npm login
```

## Setting up your package

All private packages are scoped.

Scopes are a new feature of npm. If a package's name begins with `@` , then it is a scoped package. The scope is everything in between the `@` and the slash.

`@scope/project-name`

When you sign up for private modules as an individual user, your scope is your username.

`@username/project-name`

If you use `npm init` to initialize your packages, you can pass in your scope like this:



```
npm init --scope=<your_scope>
```

If you use the same scope most of the time, you'll probably want to set it in your default configuration instead.

```
npm config set scope <your_scope>
```

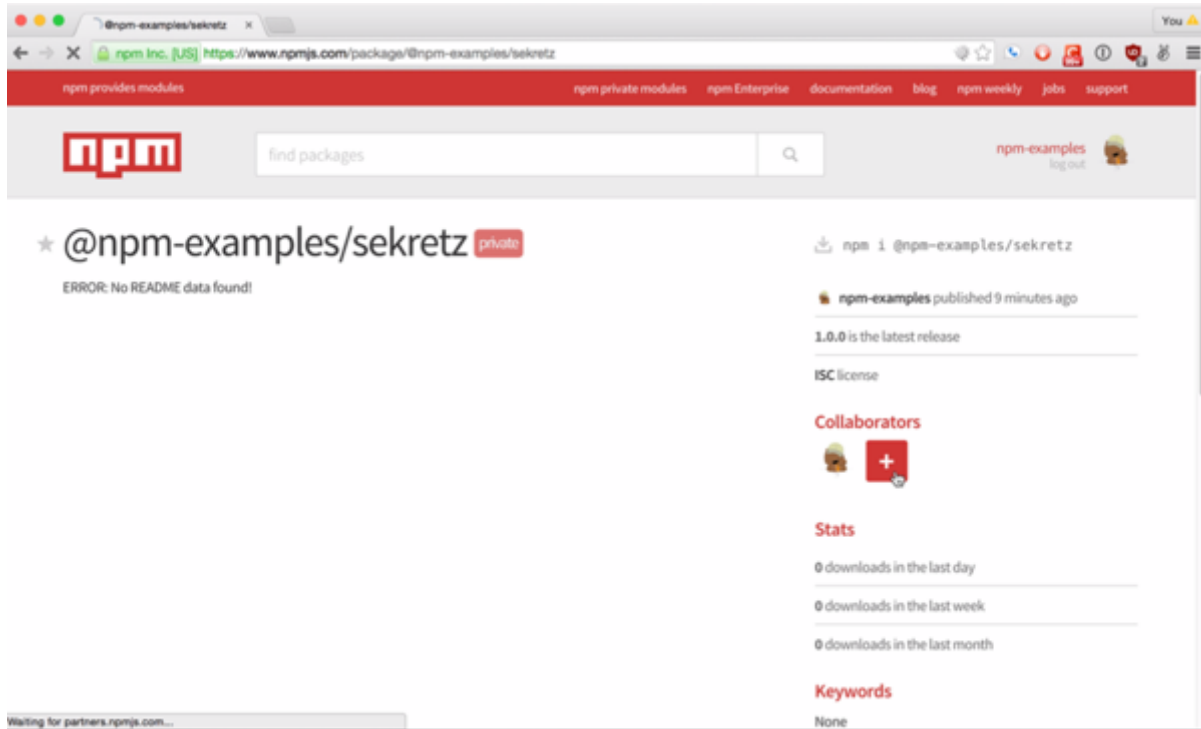
## Publishing your package

Publishing your package is easy.

```
npm publish
```

By default, scoped packages are published as private. You can read more about this in the [scopes documentation](#).

Once it's published, you should see it on the website with a private flag.



## Giving access to others

If you want to give access to someone, they need to be subscribed to private modules as well. Once they are, you can give them read or read-write access.

You can control access to the package on the access page. To get to the page, click on the Collaborators link or the plus button.

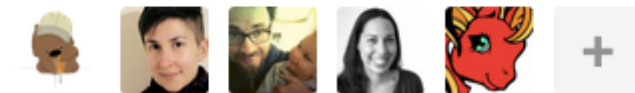
 `npm i @npm-examples/private-package`

 **npm-examples** published 4 days ago

**1.0.0** is the latest release

ISC license



## Collaborators



Add collaborators by entering the username and hitting enter.

# @npm-examples/private-package

**PRIVATE**

	<b>npm-examples</b>	read-only	read-write	✕
	<b>linclark</b>	read-only	read-write	✕
<input type="text" value="Add a collaborator..."/>				

You can also add collaborators on the command line:

```
npm owner add <user> <package name>
```

## Installing private modules

To install a private module, you must have access to the package. Then you can use install with the scoped package name.

```
npm install @scope/project-name
```

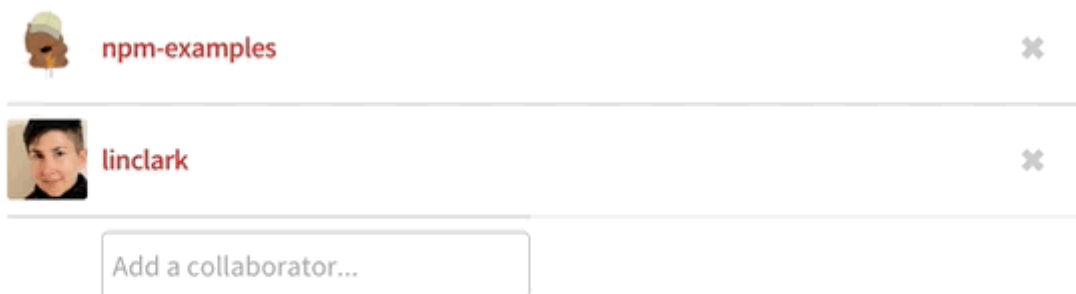
You also use the scoped package name when requiring it.

```
var project = require('@scope/project-name')
```

# Switching from private to public

All scoped packages default to private. This ensures that you don't make something public by accident. You can change this on the access page.

## @npm-examples/private-package



You can also manage package access via the command line:

```
npm access restricted <package_name>
```

The package will be removed from listings on the site within a few minutes of making it private.

---

Last modified November 04, 2016    Found a typo? Send a [pull request!](#)

## Downloading modules to CI/deployment servers

If you are using deployment servers or testing with CI servers, you'll need a way to download your private modules to those servers. To do this, you can set up an [`.npmrc`](#) file which will authenticate your server with npm.

## Getting an authentication token

One of the things that has changed in npm is that we now use auth tokens to authenticate in the CLI. To generate an auth token, you can log in on any machine. You'll end up with a line in your [`.npmrc`](#) file that looks like this:

```
//registry.npmjs.org/:_authToken=00000000-0000-0000-0000-000000000000
```

The token is not derived from your password, but changing your password will invalidate all tokens. The token will be valid until the password is changed. You can also invalidate a single token by logging out on a machine that is logged in with that token.

## Setting up environment variables

To make this more secure when pushing it up to the server, you can set this token as an environment variable on the server. For example, in Heroku you would do this:

```
heroku config:set NPM_TOKEN=00000000-0000-0000-0000-000000000000 --app=application_name
```

You will also need to add this to your environment variables on your development machine. In OSX or Linux, you would add this line to your `~/.profile` :

```
export NPM_TOKEN="00000000-0000-0000-0000-000000000000"
```

and then refresh your environment variables:

```
source ~/.profile
```

## Checking in your `.npmrc`

Then you can check in the `.npmrc` file, replacing your token with the environment variable.

```
//registry.npmjs.org/:_authToken=${NPM_TOKEN}
```

---

Last modified December 29, 2015

Found a typo? Send a [pull request!](#)

## Docker and private modules

If you've read through [Working with private modules](#), you'll know that in order to use private modules, you need to be [logged in](#) to npm via the npm CLI.

If you're using npm private modules in an environment where you're not directly able to log in, such as inside a [CI Server](#) or a [Docker](#) container, you'll need to get and export an npm token as an environment variable. That token should look like `NPM_TOKEN=00000000-0000-0000-0000-000000000000`.

The [Getting an Authentication Token](#) should help you generate that token.

If this is the workflow you need, please read the [CI Server Config doc](#). If that works with your system then perfect.

If it doesn't, here we'll look at the problems with this workflow when running `npm install` inside a Docker container.

## Runtime Variables

If you had the following Dockerfile:

```
FROM risingstack/alpine:3.3-v4.3.1-3.0.1
```

```
COPY package.json package.json
```

```
RUN npm install
```

```
# Add your source files
```

```
COPY . .
```

```
CMD npm start
```

Which will use the RisingStack [Alpine Node.JS Docker image](#), copy the `package.json` into our container, installs dependencies, copies the source files and runs the start command as specified in the `package.json`.

In order to install private packages, you may think that we could just add a line before we run `npm install`, using the [ENV parameter](#):

```
ENV NPM_TOKEN=00000000-0000-0000-0000-000000000000
```

However this doesn't work as you would expect, because you want the npm install to occur when you run `docker build`, and in this instance, `ENV` variables aren't used, they are set for runtime only.

## Build-time variables

We have to take advantage of a different way of passing environment variables to Docker, available since Docker 1.9. We'll use the slightly confusingly named [ARG parameter](#).

A complete example that will allow us to use `--build-arg` to pass in our NPM\_TOKEN requires adding a `.npmrc` file to the project. That file should contain the following content:

```
//registry.npmjs.org/:_authToken=${NPM_TOKEN}
```

The Dockerfile that takes advantage of this has a few more lines in it than our example earlier that allows us to use the `.npmrc` file and the `ARG` parameter.

```
FROM risingstack/alpine:3.3-v4.3.1-3.0.1
```

```
ARG NPM_TOKEN
COPY .npmrc .npmrc
COPY package.json package.json
RUN npm install
RUN rm -f .npmrc

# Add your source files
COPY . .
CMD npm start
```

This adds the expected `ARG NPM_TOKEN`, but also copies the `.npmrc` file, and removes it when npm install completes.

To build the image using this Dockerfile and the token, you can run the following (note the `.` at the end to give docker build the current directory as an argument):

```
docker build --build-arg NPM_TOKEN=${NPM_TOKEN} .
```

This will take your current `NPM_TOKEN` environment variable, and will build the docker image using it, so you can run `npm install` inside your container as the current logged in user!

Note: Even if you delete the `.npmrc` file, it'll be kept in the commit history - to clean your secret up entirely make sure to squash them.

---

Last modified June 17, 2016

Found a typo? Send a [pull request!](#)

## What are Organizations?

npm Organizations allow you to manage and monitor access to both new and pre-existing public and private packages through the use of teams.

A great way to think about Organizations is that they are the umbrella structure that allows you to create teams and then grant package access to those teams.

These docs can be seen as being separated into 2 sections: people and packages.

## Managing People ( `npm team` )

- [Creating an Organization](#)
- [Definition of Organizational Roles](#)
- [Teams and Team Member Access](#)

## Managing Package Access ( `npm access` )

An Organization can collaborate on 2 types of packages:

- [New packages scoped to the Organization](#)
- [Pre-existing public and private packages](#)

Additionally, all team members have the ability to [monitor access to packages](#).

## CLI Documentation

For full documentation on the CLI commands associated with this feature:

- [npm team](#)
- [npm access](#)

---

Last modified November 29, 2015

Found a typo? Send a [pull request!](#)

## Setting up an Organization

[Organizations](#) and [Organization membership](#) are created in the [npm web interface](#).

## Creating an Organization

In order to create an Organization, you must be logged in as a npm user with a verified email address. To create an npm user, [click here](#). There are 2 ways to create an organization:

- [New Scope](#)
- [Pre-existing User Scope](#)

Create an Organization with a new scope:

1. Log in to <http://www.npmjs.com/>
2. Visit <https://www.npmjs.com/org/create>
3. Click the big red button "Create an Organization"

## Organization Dashboard

Once you've created an Organization, you can perform a wide variety of tasks on your Organization Dashboard.

Your Organization Dashboard is located:

<https://www.npmjs.com/org/<org>>

...where **<org>** is the name of your Organization.

## Adding Members to an Organization

By default, your Organization is set up with a [developers team](#). Whenever you add a new member to your Organization, they are automatically added to the developers team.

You may delete the developers team. If you do, newly added Organization members will not be added to any teams by default.

For more information about the developers team, see [Developers Team](#)

To add a member to your organization, you add them by their npm username via the [Organization Dashboard](#).

## Creating Team Admins

As the creator of the Organization you are granted the role of [Super Admin](#).

For more information about the [Super Admin](#) and [Team Admin](#) roles, checkout the [Roles](#) documentation.

## Migrating an existing username to an Org

Many users have already registered an npm user with the @scope they want to use for their org. If you attempt to register an org with a scope already in use, and you are already logged in as that user, you will be prompted to automatically migrate that user to an org.

Once your @scope is owned by an org, **you can no longer log in as your former username**. Orgs are not users and do not have usernames and passwords. During migration, you will be prompted to pick a new username. This new user will have the same password as your old user, but all packages that belonged to your old user will now belong to the org. Your new user will have Super-Admin privileges to the org.

## Roles

Organizations are first and foremost a way to manage access, roles and responsibilities. Organizations offer 3 types of roles, and also have an interface with the general public:

### . Super-Admin

The user who creates the Organization is automatically set as the Super Admin.

- can see/do everything regarding their org
- can pay for the org
- can [add users to the org \(team-admin or developer\)](#)

Currently, only one Super Admin is allowed and the Super Admin User cannot be changed. New versions of our Orgs product will make this possible.

### . Team-Admin

Team Admins are set by the Super Admin in the [website interface](#). There can be  $\geq 0$  Team Admins.

- can [see teams](#)
- can add a [new org-scoped package](#)
- can add a [pre-existing package](#)
- cannot pay
- cannot add/remove users to/from org
- can [add/remove users to/from teams](#)

### . Member

Members are added to teams by the Organizations Super Admin or Team Admin.

- can [see the teams they're on](#)
- can [see the packages associated with those teams](#)
- can [add a new org-scoped package](#)
- cannot pay
- cannot add users

### . General Public

While not associated with the Organization, the general public has some ability to interact with Organizations.

- can see the org exists
- can see public packages in the org's domain
- cannot see private packages
- cannot see members
- cannot do anything for the org (manage members, teams, packages, billing, etc)

When you first create an Organization, a team called **developers** is created.

The **developers** team is a **special team**. While it can be deleted [if you so choose](#), by default it acts as a "catch-all" team. This means:

- new Organization members will be added to the **developers** team by default
- the **developers** team has write access to all new Organization-scope package publishes

The effects of deleting the team are [covered below](#).

## Removing the developers team

You may delete the developers team. If you do, newly added Organization members will not be added to any teams by default. Additionally, you will not be able to see all users in your org from the CLI, as one can only view the members of a team via the CLI.

You should also note that upon publish, in the absence of a **developers** team, it is difficult to determine who should be set as maintainers of that package. npm will do its best to fallback to another Organization team that the publisher is a member of. This is not predictable.

## Reinstating the developers team

If you've removed the developers team, but now want it back, you can reinstate it by creating a new team called **developers** (case sensitive!). You will need to add all current members of the Organization to the new **developers** team, but, going forward all newly added Organization members will be automatically added to the new **developers** team.

---

Last modified February 29, 2016

Found a typo? Send a [pull request!](#)

## Teams

The key to managing access to packages via Organizations is Teams.

## What are Teams?

Teams are sets of users that have access to a certain scope within the Organization.

In order to create teams and manage team membership, you must be a [Super Admin](#) or [Team Admin](#) under the given organization. Listing teams and team memberships may be done by any member of the organization.

Organization creation and management of [Team Admin](#) and [Team Member](#) roles is done through the web interface.

## Creating Teams

A [Super Admin](#) or [Team Admin](#) has the ability to create a team. To create a team one can type:

```
> npm team create <org:team>
```

...where **<org:team>** is the name of the Organization, followed by the name of the new team.

For example, to create a team called **wombats** in the **@npminc** Organization, a [Super Admin](#) or [Team Admin](#) would type:

```
> npm team create npminc:wombats
```

You can check that you created the team successfully by listing the teams in your Organization. You can do that by typing:

```
> npm team ls <org>
```

or by visiting the [Organization Dashboard](#) in the web interface.

## Adding Users to a Team



Once you've created a team you'll want to add users to it. To do so a [Super Admin](#) or [Team Admin](#) can type:

```
> npm team add <org:team> <user>
```

...where [org:team](#) is the name of the Organization, followed by the name of the team and is the npm username of the user you'd like to make a member of the team.

For example, to make the npm user `ag_dubs` a member of the `@npminc` organization's `wombats` team:

```
> npm team add npminc:wombats ag_dubs
```

To check if you've added a user successfully, you can list all the users on a particular team. To do so, type:

```
> npm team ls <org:team>
```

## Removing a User from a Team

```
> npm team rm <org:team> <user>
```

## Listing Teams and Team Members

### List an Organization's Teams

```
> npm team ls <org>
```

### List a Team's Members

```
> npm team ls <org:team>
```

## CLI Documentation

For detailed information on the `team` command, check out the CLI documentation [here](#).

---

Last modified January 12, 2017

Found a typo? Send a [pull request!](#)

## Sponsorship

## Sponsorship Types

# @ag-org








0 packages

1 team


4 members

payment info

## 4 active members

collaborators	role	status		action
 <b>ag_dubs</b> ashley williams	super-admin	unpaid	<input checked="" type="checkbox"/>	<b>paid</b>
 <b>CarlSagan</b> Carl Sagan	developer	unpaid	<input type="checkbox"/>	<b>paid</b> 
 <b>EmmaGoldman</b> Emma Goldman	developer	unpaid	<input type="checkbox"/>	<b>paid</b> 
 <b>JacquesDerrida</b> Jacques Derrida	developer	unpaid	<input checked="" type="checkbox"/>	<b>paid</b> 
<div>Invite a Member</div> <div>Member <input checked="" type="radio"/> Team Admin <input type="radio"/></div> <div>add member</div>				

Let's say we have an Organization, `@ag_org` . This Organization was created by user `@ag_dubs` , and therefore she is the Super Admin .

 **ag\_dubs**  
ashley williams

super-admin

unpaid ☒ **paid**

Being a Super Admin, she adds 3 members to her team:

- Jacques Derrida
- Carl Sagan
- Emma Goldman

There are three types of Sponsorship that can occur:

- Paid by Current Organization
- Paid by Self or Another Organization
- Not Paid

### Paid by Current Organization

When Super Admin, `@ag_dubs` , added `JacquesDerrida` to the Organization, `JacquesDerrida` did not already belong to an organization nor did they have a subscription to private packages.

By default, when the Super Admin added him to the Organization, `JacquesDerrida` was set as a member of the Organization, **paid by the current organization**. This appears in the UI like this:

 **JacquesDerrida**  
Jacques Derrida

developer

unpaid ☒ **paid** 

- Because `JacquesDerrida` is a **developer** in the org, they can:
  - Be added to any/all of the Organization's teams,
  - See the teams they are on,
  - See the other members of the team,
  - See the packages (both public and private) those teams grant access to,
- Because `JacquesDerrida` is sponsored, they can:
  - Install, publish, and unpublish the private packages to which they have access

**JacquesDerrida** cannot:

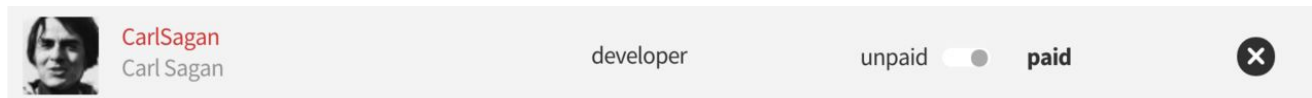
- See all of the Organization's teams
- See all members of the Organization

## Paid by Self or Another Organization

### • [Paid for by Another Organization](#)

When Super Admin, **@ag\_dubs** , added **CarlSagan** to the Organization, **CarlSagan** already belonged to another Organization ( **@nasa-org** , duh).

By default, when **CarlSagan** was added to the Organization, he was set as a member of the Organization, **paid by another scope**. This appears in the UI like this:



As a result, **CarlSagan** has the same permissions as **JacquesDerrida** , above.

### • [Paid for by Self](#)

As a subscriber to Private Packages, you can understand your sponsorship as "sponsoring yourself". As a result, a subscriber to private packages would have had the same default behavior as occurred for **CarlSagan** , i.e., the previous sponsorship would trump the possibility of a new Organization sponsorship. A subscriber to private packages will appear in the Organization dashboard as someone who is **paid** but **not by the current org** . This appears in the UI the same as above:



### • [Changing Sponsorship](#)

If a user is a subscriber to private packages, this sponsorship scope will trump all other potential sponsorships. If you would like to change this, i.e., offer sponsorship to a user who already has another sponsorship (org or private pkgs), please contact [support@npmjs.com](mailto:support@npmjs.com).

### • Not Paid

**EmmaGoldman** , at the time that **@ag\_dubs** added them to the **@ag-org** Organization, did not subscribe to private packages nor did they belong to another Organization. This means that they did not have any previous sponsorships.

Like **JacquesDerrida** , **EmmaGoldman** was set as **paid by the current organization**, **@ag-org** by default. However, Super Admin **@ag\_dubs** opted to cancel **@ag-org** 's sponsorship of **EmmaGoldman** . This status appears in the UI like this:



- Because **EmmaGoldman** is a **developer** in the org, they can:
  - Be added to any/all of the Organization's teams,
  - See the teams they are on,
  - See the other members of the team,
  - See the packages (both public and private) those teams grant access to,
- Because **EmmaGoldman** is *not* sponsored, they *cannot*:
  - Install, publish, and unpublish the private packages to which they have access

**EmmaGoldman** can:

- Collaborate on and publish public, scoped or unscoped packages, that their team membership grants them access to

**EmmaGoldman** cannot:

- See all of the Organization's teams

- See all members of the Organization
- Collaborate on and publish any private packages, even if their team membership would otherwise grant them access

---

Last modified December 29, 2015

Found a typo? Send a [pull request!](#)

## Scoping a Package to your Organization

Once you have an Organization set up, you'll want to scope packages to that Organization.

Users with [Super Admin](#), [Team Admin](#), and [Member](#) roles can perform this action.

## Scoping a New Package

To do this, run these commands in the root directory of your package:

```
> npm init --scope=<org>
> npm publish
```

... where **<org>** is the name of your Organization.

## Setting your Organization as Default Scope

If you are using Organizations, there is a good chance that you'll be using the Organization scope regularly.

To save yourself some typing, you can set your Organization as your default scope:

```
npm config set scope <org>
```

... where **<org>** is the name of your Organization.

---

Last modified December 29, 2015

Found a typo? Send a [pull request!](#)

## Managing Organization Package Access

Once you have scoped a package to your Organization, users with [Super Admin](#) or [Team Admin](#) roles in your Organization can [grant](#), [revoke](#), and [monitor](#) team access to that package.

## Access Levels

There are two levels of access you can provide:

- read-only: can use the package, e.g. `npm install`
- read-write: can update the package, e.g. `npm publish`

## Granting Access

To grant access to a team, a [Team Admin](#) can type:

```
> npm access grant <read-only|read-write> <org:team> [<package>]
```

The `grant` command takes 3 arguments, in order:

- access level: `read-only` or `read-write`
- scope: `<org:team>`, e.g. `npm:inc:wombats`

- package: the name of a package The package must be:
  - [scoped to the Organization](#), or
  - be scoped package that a [Super Admin](#) or [Team Admin](#) has write-access to Package name is optional if you are executing the command in a directory with a [package.json](#) .

For example, to grant read-write access the `npm-docs` package to the `@npminc` org's `wombats` team, a user who:

- is a [Super Admin](#) or [Team Admin](#) for the Organization
- has write access to the `npm-docs` package

...would do the following:

```
> npm access grant read-write npminc:wombats npm-docs
```

## Revoking Access

To revoke team access to a package, a [Team Admin](#) can type:

```
> npm access revoke <org:team> [<package>]
```

Again, the `package` argument is optional if this command is executed in a directory containing a [package.json](#) .

## Monitor Access

You can check whether you have successfully granted or revoked team access to a package using the `npm access ls-packages` and `npm access ls-collaborators` command.

### View a Team Member's Package Access

```
npm access ls-packages <org> <user>
```

### View a Team's Package Access

```
npm access ls-packages <org:team>
```

### List Teams with Access to a Package

```
npm access ls-collaborators <pkg>
```

---

Last modified December 29, 2015

Found a typo? Send a [pull request!](#)

## Manage Team Access To Previously Existing Packages

## Restrictions

Currently, it is not possible to change the scope of a pre-existing public or private, scoped or not, package to an Organization.

Specifically, given a private, scoped package `@ag_dubs/foo` , there is currently no way to make that exact package scoped to the Organization, `@ag_org` , i.e. `@ag_org/foo` without creating a new package.

However, Organization members who are either a

- [Super Admin](#), or
- [Team Admin](#)

that are also:

- an admin member of the package's org, for org-scoped packages
- the user, for user-scoped packages

and, as of `npm@3.5.0/npm@2.14.12` :

- the last publisher on a public package

... are able to grant Organization team access to packages that are not scoped within the Organization.

### Examples

- **meow-org** Super Admin, Irina, is also a Team Admin for **pizza-org**. Irina can grant the **meow-org/cyborgs** team access to the **pizza-org/pepperoni** package.
- **puppyco/corgis** Team Admin, Lewis, has a personal private package, **@lewis/corgis**. Lewis can grant the **puppyco/corgis** team access to his **@lewis/corgis** package.
- **cactus-inc** Super Admin, Corey, was also the last person to publish the public package, **bdaypresent**. Corey can grant the **cactus-inc/friends** team access to the **bdaypresent** package.

(\*yup. this is weird. we know.)

Note: It is possible to migrate a User scope to an Organization scope. For more information on that check out the [Migrating a Current User Scope to an Org](#) in the [Creating an Org documentation](#).

## Granting Team Access to a Package

So, let's say you have a package **@ag\_dubs/foo** that you would like to collaborate on within the Organization **@ag\_org**.

First, ensure that you have the correct permissions. The user must:

- Be a [Super Admin](#) or [Team Admin](#) in the Organization
- Be an admin of the package, **@ag\_dubs/foo**

Then, you can [grant team access to a package](#), as though it were scoped to the Organization:

```
> > npm access grant <read-only|read-write> <org:team> @ag_dubs/foo
```

---

Last modified December 29, 2015

Found a typo? Send a [pull request!](#)

## npm-coding-style

### Description

npm's coding style is a bit unconventional. It is not different for difference's sake, but rather a carefully crafted style that is designed to reduce visual clutter and make bugs more apparent.

If you want to contribute to npm (which is very encouraged), you should make your code conform to npm's style.

Note: this concerns npm's code not the specific packages that you can download from the npm registry.

### Line Length

Keep lines shorter than 80 characters. It's better for lines to be too short than to be too long. Break up long lists, objects, and other statements onto multiple lines.

### Indentation

Two-spaces. Tabs are better, but they look like hell in web browsers (and on GitHub), and node uses 2 spaces, so that's that.

Configure your editor appropriately.

## Curly braces

Curly braces belong on the same line as the thing that necessitates them.

Bad:

```
function ()  
{
```

Good:

```
function () {
```

If a block needs to wrap to the next line, use a curly brace. Don't use it if it doesn't.

Bad:

```
if (foo) { bar() }  
while (foo)  
  bar()
```

Good:

```
if (foo) bar()  
while (foo) {  
  bar()  
}
```

## Semicolons

Don't use them except in four situations:

- **for (;;)** loops. They're actually required.
- null loops like: **while (something) ;** (But you'd better have a good reason for doing that.)
- **case 'foo': doSomething(); break**
- In front of a leading **(** or **[** at the start of the line. This prevents the expression from being interpreted as a function call or property access, respectively.

Some examples of good semicolon usage:

```
;(x || y).doSomething()  
[a, b, c].forEach(doSomething)  
for (var i = 0; i < 10; i++) {  
  switch (state) {  
    case 'begin': start(); continue  
    case 'end': finish(); break  
    default: throw new Error('unknown state')  
  }  
  end()  
}
```

Note that starting lines with **-** and **+** also should be prefixed with a semicolon, but this is much less common.

## Comma First

If there is a list of things separated by commas, and it wraps across multiple lines, put the comma at the start of the next line, directly below the token that starts the list. Put the final token in the list on a line by itself. For example:

```
var magicWords = [  
  'abracadabra'  
  , 'gesundheit'
```

```
    , 'ventrilo'
  ]
  , spells = { 'fireball' : function () { setOnFire() }
    , 'water' : function () { putOut() }
  }
  , a = 1
  , b = 'abc'
  , etc
  , somethingElse
```

## Quotes

Use single quotes for strings except to avoid escaping.

Bad:

```
var notOk = "Just double quotes"
```

Good:

```
var ok = 'String contains "double" quotes'
var alsoOk = "String contains 'single' quotes or apostrophe"
```

## Whitespace

Put a single space in front of ( for anything other than a function call. Also use a single space wherever it makes things more readable.

Don't leave trailing whitespace at the end of lines. Don't indent empty lines. Don't use more spaces than are helpful.

## Functions

Use named functions. They make stack traces a lot easier to read.

## Callbacks, Sync/async Style

Use the asynchronous/non-blocking versions of things as much as possible. It might make more sense for npm to use the synchronous fs APIs, but this way, the fs and http and child process stuff all uses the same callback-passing methodology.

The callback should always be the last argument in the list. Its first argument is the Error or null.

Be very careful never to ever ever throw anything. It's worse than useless. Just send the error message back as the first argument to the callback.

## Errors

Always create a new Error object with your message. Don't just return a string message to the callback. Stack traces are handy.

## Logging

Logging is done using the [npmlog](#) utility.

Please clean up logs when they are no longer helpful. In particular, logging the same object over and over again is not helpful. Logs should report what's happening so that it's easier to track down where a fault occurs.

Use appropriate log levels. See [npm-config](#) and search for "loglevel".

## Case, naming, etc.



Use `lowerCamelCase` for multiword identifiers when they refer to objects, functions, methods, properties, or anything not specified in this section.

Use `UpperCamelCase` for class names (things that you'd pass to "new").

Use `all-lower-hyphen-css-case` for multiword filenames and config keys.

Use named functions. They make stack traces easier to follow.

Use `CAPS_SNAKE_CASE` for constants, things that should never change and are rarely used.

Use a single uppercase letter for function names where the function would normally be anonymous, but needs to call itself recursively. It makes it clear that it's a "throwaway" function.

## null, undefined, false, 0

Boolean variables and functions should always be either `true` or `false`. Don't set it to 0 unless it's supposed to be a number.

When something is intentionally missing or removed, set it to `null`.

Don't set things to `undefined`. Reserve that value to mean "not yet set to anything."

Boolean objects are verboten.

## See Also

- [npm-developers](#)
- [npm-faq](#)
- [npm](#)

---

Last modified January 08, 2016

Found a typo? Send a [pull request!](#)

## npm-config

### Description

npm gets its configuration values from the following sources, sorted by priority:

#### Command Line Flags

Putting `--foo bar` on the command line sets the `foo` configuration parameter to `"bar"`. A `--` argument tells the cli parser to stop reading flags. A `--flag` parameter that is at the *end* of the command will be given the value of `true`.

#### Environment Variables

Any environment variables that start with `npm_config_` will be interpreted as a configuration parameter. For example, putting `npm_config_foo=bar` in your environment will set the `foo` configuration parameter to `bar`. Any environment configurations that are not given a value will be given the value of `true`. Config values are case-insensitive, so `NPM_CONFIG_FOO=bar` will work the same.

#### npmrc Files

The four relevant files are:

- per-project config file (/path/to/my/project/.npmrc)
- per-user config file (~/.npmrc)
- global config file (\$PREFIX/etc/npmrc)
- npm builtin config file (/path/to/npm/npmrc)

See [npmrc](#) for more details.

## Default Configs

Run `npm config ls -l` to see a set of configuration parameters that are internal to npm, and are defaults if nothing else is specified.

# Shorthands and Other CLI Niceties

The following shorthands are parsed on the command-line:

- `-v` : `--version`
- `-h` , `-?` , `--help` , `-H` : `--usage`
- `-s` , `--silent` : `--loglevel silent`
- `-q` , `--quiet` : `--loglevel warn`
- `-d` : `--loglevel info`
- `-dd` , `--verbose` : `--loglevel verbose`
- `-ddd` : `--loglevel silly`
- `-g` : `--global`
- `-C` : `--prefix`
- `-l` : `--long`
- `-m` : `--message`
- `-p` , `--porcelain` : `--parseable`
- `-reg` : `--registry`
- `-f` : `--force`
- `-desc` : `--description`
- `-S` : `--save`
- `-D` : `--save-dev`
- `-O` : `--save-optional`
- `-B` : `--save-bundle`
- `-E` : `--save-exact`
- `-y` : `--yes`
- `-n` : `--yes false`
- `ll` and `la` commands: `ls --long`

If the specified configuration param resolves unambiguously to a known configuration parameter, then it is expanded to that configuration parameter. For example:

```
npm ls --par
# same as:
npm ls --parseable
```

If multiple single-character shorthands are strung together, and the resulting combination is unambiguously not some other configuration param, then it is expanded to its various component pieces. For example:

```
npm ls -gpld
# same as:
npm ls --global --parseable --long --loglevel info
```

## Per-Package Config Settings

When running scripts (see [npm-scripts](#)) the package.json "config" keys are overwritten in the environment if there is a config param of `<name>[@<version>]:<key>`. For example, if the package.json has this:

```
{ "name" : "foo"
, "config" : { "port" : "8080" }
, "scripts" : { "start" : "node server.js" } }
```

and the server.js is this:

```
http.createServer(...).listen(process.env.npm_package_config_port)
```

then the user could change the behavior by doing:

```
npm config set foo:port 80
```

See [package.json](#) for more information.

## Config Settings

### access

- Default: **restricted**
- Type: Access

When publishing scoped packages, the access level defaults to **restricted** . If you want your scoped package to be publicly viewable (and installable) set **--access=public** . The only valid values for **access** are **public** and **restricted** . Unscoped packages *always* have an access level of **public** .

### always-auth

- Default: false
- Type: Boolean

Force npm to always require authentication when accessing the registry, even for **GET** requests.

### also

- Default: null
- Type: String

When "dev" or "development" and running local **npm shrinkwrap** , **npm outdated** , or **npm update** , is an alias for **--dev** .

### bin-links

- Default: **true**
- Type: Boolean

Tells npm to create symlinks (or **.cmd** shims on Windows) for package executables.

Set to false to have it not do this. This can be used to work around the fact that some file systems don't support symlinks, even on ostensibly Unix systems.

### browser

- Default: OS X: **"open"** , Windows: **"start"** , Others: **"xdg-open"**
- Type: String

The browser that is called by the **npm docs** command to open websites.

### ca

- Default: The npm CA certificate
- Type: String, Array or null

The Certificate Authority signing certificate that is trusted for SSL connections to the registry. Values should be in PEM format with newlines replaced by the string **"\n"**. For example:

```
ca="-----BEGIN CERTIFICATE-----\nXXXX\nXXXX\n-----END CERTIFICATE-----"
```

Set to **null** to only allow "known" registrars, or to a specific CA cert to trust only that specific signing authority.

Multiple CAs can be trusted by specifying an array of certificates:

```
ca[ ]="..."
ca[ ]="..."
```

See also the **strict-ssl** config.

## cafile

- Default: **null**
- Type: path

A path to a file containing one or multiple Certificate Authority signing certificates. Similar to the `ca` setting, but allows for multiple CA's, as well as for the CA information to be stored in a file on disk.

## cache

- Default: Windows: `%AppData%\npm-cache` , Posix: `~/.npm`
- Type: path

The location of npm's cache directory. See [npm-cache](#)

## cache-lock-stale

- Default: 60000 (1 minute)
- Type: Number

The number of ms before cache folder lockfiles are considered stale.

## cache-lock-retries

- Default: 10
- Type: Number

Number of times to retry to acquire a lock on cache folder lockfiles.

## cache-lock-wait

- Default: 10000 (10 seconds)
- Type: Number

Number of ms to wait for cache lock files to expire.

## cache-max

- Default: Infinity
- Type: Number

The maximum time (in seconds) to keep items in the registry cache before re-checking against the registry.

Note that no purging is done unless the `npm cache clean` command is explicitly used, and that only GET requests use the cache.

## cache-min

- Default: 10
- Type: Number

The minimum time (in seconds) to keep items in the registry cache before re-checking against the registry.

Note that no purging is done unless the `npm cache clean` command is explicitly used, and that only GET requests use the cache.

## cert

- Default: **null**
- Type: String

A client certificate to pass when accessing the registry.

## color

- Default: true
- Type: Boolean or **"always"**

If false, never shows colors. If **"always"** then always shows colors. If true, then only prints color codes for tty file descriptors.

## depth

- Default: Infinity
- Type: Number

The depth to go when recursing directories for `npm ls`, `npm cache ls`, and `npm outdated`.

For `npm outdated`, a setting of `Infinity` will be treated as `0` since that gives more useful information. To show the outdated status of all packages and dependents, use a large integer value, e.g., `npm outdated --depth 9999`

## description

- Default: true
- Type: Boolean

Show the description in `npm search`

## dev

- Default: false
- Type: Boolean

Install `dev-dependencies` along with packages.

Note that `dev-dependencies` are also installed if the `npm` flag is set.

## dry-run

- Default: false
- Type: Boolean

Indicates that you don't want npm to make any changes and that it should only report what it would have done. This can be passed into any of the commands that modify your local installation, eg, `install`, `update`, `dedupe`, `uninstall`. This is NOT currently honored by network related commands, eg `dist-tags`, `owner`, `publish`, etc.

## editor

- Default: `EDITOR` environment variable if set, or `"vi"` on Posix, or `"notepad"` on Windows.
- Type: path

The command to run for `npm edit` or `npm config edit`.

## engine-strict

- Default: false
- Type: Boolean

If set to true, then npm will stubbornly refuse to install (or even consider installing) any package that claims to not be compatible with the current Node.js version.

## force

- Default: false
- Type: Boolean

Makes various commands more forceful.

- lifecycle script failure does not block progress.
- publishing clobbers previously published versions.
- skips cache when requesting from the registry.
- prevents checks against clobbering non-npm files.

## fetch-retries

- Default: 2
- Type: Number

The "retries" config for the `retry` module to use when fetching packages from the registry.

## fetch-retry-factor

- Default: 10
- Type: Number

The "factor" config for the **retry** module to use when fetching packages.

### fetch-retry-mintimeout

- Default: 10000 (10 seconds)
- Type: Number

The "minTimeout" config for the **retry** module to use when fetching packages.

### fetch-retry-maxtimeout

- Default: 60000 (1 minute)
- Type: Number

The "maxTimeout" config for the **retry** module to use when fetching packages.

### git

- Default: **"git"**
- Type: String

The command to use for git commands. If git is installed on the computer, but is not in the **PATH**, then set this to the full path to the git binary.

### git-tag-version

- Default: **true**
- Type: Boolean

Tag the commit when using the **npm version** command.

### global

- Default: false
- Type: Boolean

Operates in "global" mode, so that packages are installed into the **prefix** folder instead of the current working directory. See [npm-folders](#) for more on the differences in behavior.

- packages are installed into the **{prefix}/lib/node\_modules** folder, instead of the current working directory.
- bin files are linked to **{prefix}/bin**
- man pages are linked to **{prefix}/share/man**

### globalconfig

- Default: **{prefix}/etc/npmrc**
- Type: path

The config file to read for global config options.

### global-style

- Default: false
- Type: Boolean

Causes npm to install the package into your local **node\_modules** folder with the same layout it uses with the global **node\_modules** folder. Only your direct dependencies will show in **node\_modules** and everything they depend on will be flattened in their **node\_modules** folders. This obviously will eliminate some deduping. If used with **legacy-bundling**, **legacy-bundling** will be preferred.

### group

- Default: GID of the current process
- Type: String or Number

The group to use when running package scripts in global mode as the root user.

## heading

- Default: `"npm"`
- Type: String

The string that starts all the debugging log output.

## https-proxy

- Default: null
- Type: url

A proxy to use for outgoing https requests. If the `HTTPS_PROXY` or `https_proxy` or `HTTP_PROXY` or `http_proxy` environment variables are set, proxy settings will be honored by the underlying `request` library.

## if-present

- Default: false
- Type: Boolean

If true, npm will not exit with an error code when `run-script` is invoked for a script that isn't defined in the `scripts` section of `package.json`. This option can be used when it's desirable to optionally run a script when it's present and fail if the script fails. This is useful, for example, when running scripts that may only apply for some builds in an otherwise generic CI setup.

## ignore-scripts

- Default: false
- Type: Boolean

If true, npm does not run scripts specified in package.json files.

## init-module

- Default: `~/.npm-init.js`
- Type: path

A module that will be loaded by the `npm init` command. See the documentation for the [init-package-json](#) module for more information, or [npm-init](#).

## init-author-name

- Default: `""`
- Type: String

The value `npm init` should use by default for the package author's name.

## init-author-email

- Default: `""`
- Type: String

The value `npm init` should use by default for the package author's email.

## init-author-url

- Default: `""`
- Type: String

The value `npm init` should use by default for the package author's homepage.

## init-license

- Default: `"ISC"`
- Type: String

The value `npm init` should use by default for the package license.

## init-version

- Default: `"1.0.0"`

- Type: semver

The value that `npm init` should use by default for the package version number, if not already set in package.json.

## json

- Default: false
- Type: Boolean

Whether or not to output JSON data, rather than the normal output.

This feature is currently experimental, and the output data structures for many commands is either not implemented in JSON yet, or subject to change. Only the output from `npm ls --json` is currently valid.

## key

- Default: `null`
- Type: String

A client key to pass when accessing the registry.

## legacy-bundling

- Default: false
- Type: Boolean

Causes npm to install the package such that versions of npm prior to 1.4, such as the one included with node 0.8, can install the package. This eliminates all automatic deduping. If used with `global-style` this option will be preferred.

## link

- Default: false
- Type: Boolean

If true, then local installs will link if there is a suitable globally installed package.

Note that this means that local installs can cause things to be installed into the global space at the same time. The link is only done if one of the two conditions are met:

- The package is not already installed globally, or
- the globally installed version is identical to the version that is being installed locally.

## local-address

- Default: undefined
- Type: IP Address

The IP address of the local interface to use when making connections to the npm registry. Must be IPv4 in versions of Node prior to 0.12.

## loglevel

- Default: "warn"
- Type: String
- Values: "silent", "error", "warn", "http", "info", "verbose", "silly"

What level of logs to report. On failure, *all* logs are written to `npm-debug.log` in the current working directory.

Any logs of a higher level than the setting are shown. The default is "warn", which shows warn and error output.

## logstream

- Default: process.stderr
- Type: Stream

This is the stream that is passed to the `npmlog` module at run time.

It cannot be set from the command line, but if you are using npm programmatically, you may wish to send logs to somewhere other than stderr.

If the `color` config is set to true, then this stream will receive colored output if it is a TTY.



## long

- Default: false
- Type: Boolean

Show extended information in `npm ls` and `npm search`.

## message

- Default: "%s"
- Type: String

Commit message which is used by `npm version` when creating version commit.

Any "%s" in the message will be replaced with the version number.

## node-version

- Default: process.version
- Type: semver or false

The node version to use when checking a package's `engines` map.

## npat

- Default: false
- Type: Boolean

Run tests on installation.

## onload-script

- Default: false
- Type: path

A node module to `require()` when npm loads. Useful for programmatic usage.

## only

- Default: null
- Type: String

When "dev" or "development" and running local `npm install` without any arguments, only devDependencies (and their dependencies) are installed.

When "dev" or "development" and running local `npm ls`, `npm outdated`, or `npm update`, is an alias for `--dev`.

When "prod" or "production" and running local `npm install` without any arguments, only non-devDependencies (and their dependencies) are installed.

When "prod" or "production" and running local `npm ls`, `npm outdated`, or `npm update`, is an alias for `--production`.

## optional

- Default: true
- Type: Boolean

Attempt to install packages in the `optionalDependencies` object. Note that if these packages fail to install, the overall installation process is not aborted.

## parseable

- Default: false
- Type: Boolean

Output parseable results from commands that write to standard output.

## prefix

- Default: see [npm-folders](#)
- Type: path

The location to install global items. If set on the command line, then it forces non-global commands to run in the specified folder.

## production

- Default: false
- Type: Boolean

Set to true to run in "production" mode.

1. devDependencies are not installed at the topmost level when running local `npm install` without any arguments.
2. Set the `NODE_ENV="production"` for lifecycle scripts.

## progress

- Default: true
- Type: Boolean

When set to `true`, npm will display a progress bar during time intensive operations, if `process.stderr` is a TTY.

Set to `false` to suppress the progress bar.

## proprietary-attrs

- Default: true
- Type: Boolean

Whether or not to include proprietary extended attributes in the tarballs created by npm.

Unless you are expecting to unpack package tarballs with something other than npm -- particularly a very outdated tar implementation -- leave this as true.

## proxy

- Default: null
- Type: url

A proxy to use for outgoing http requests. If the `HTTP_PROXY` or `http_proxy` environment variables are set, proxy settings will be honored by the underlying `request` library.

## rebuild-bundle

- Default: true
- Type: Boolean

Rebuild bundled dependencies after installation.

## registry

- Default: <https://registry.npmjs.org/>
- Type: url

The base URL of the npm package registry.

## rollback

- Default: true
- Type: Boolean

Remove failed installs.

## save

- Default: false
- Type: Boolean

Save installed packages to a package.json file as dependencies.

When used with the `npm rm` command, it removes it from the `dependencies` object.

Only works if there is already a package.json file present.

## save-bundle

- Default: false

- Type: Boolean

If a package would be saved at install time by the use of `--save` , `--save-dev` , or `--save-optional` , then also put it in the `bundleDependencies` list.

When used with the `npm rm` command, it removes it from the `bundleDependencies` list.

### save-dev

- Default: false
- Type: Boolean

Save installed packages to a package.json file as `devDependencies` .

When used with the `npm rm` command, it removes it from the `devDependencies` object.

Only works if there is already a package.json file present.

### save-exact

- Default: false
- Type: Boolean

Dependencies saved to package.json using `--save` , `--save-dev` or `--save-optional` will be configured with an exact version rather than using npm's default semver range operator.

### save-optional

- Default: false
- Type: Boolean

Save installed packages to a package.json file as `optionalDependencies`.

When used with the `npm rm` command, it removes it from the `devDependencies` object.

Only works if there is already a package.json file present.

### save-prefix

- Default: '^'
- Type: String

Configure how versions of packages installed to a package.json file via `--save` or `--save-dev` get prefixed.

For example if a package has version `1.2.3` , by default its version is set to `^1.2.3` which allows minor upgrades for that package, but after `npm config set save-prefix='~'` it would be set to `~1.2.3` which only allows patch upgrades.

### scope

- Default: ""
- Type: String

Associate an operation with a scope for a scoped registry. Useful when logging in to a private registry for the first time: `npm login --scope=@organization --registry=registry.organization.com` , which will cause `@organization` to be mapped to the registry for future installation of packages specified according to the pattern `@organization/package` .

### searchopts

- Default: ""
- Type: String

Space-separated options that are always passed to search.

### searchexclude

- Default: ""
- Type: String

Space-separated options that limit the results from search.

## searchsort

- Default: "name"
- Type: String
- Values: "name", "-name", "date", "-date", "description", "-description", "keywords", "-keywords"

Indication of which field to sort search results by. Prefix with a `-` character to indicate reverse sort.

## shell

- Default: SHELL environment variable, or "bash" on Posix, or "cmd" on Windows
- Type: path

The shell to run for the `npm explore` command.

## shrinkwrap

- Default: true
- Type: Boolean

If set to false, then ignore `npm-shrinkwrap.json` files when installing.

## sign-git-tag

- Default: false
- Type: Boolean

If set to true, then the `npm version` command will tag the version using `-s` to add a signature.

Note that git requires you to have set up GPG keys in your git configs for this to work properly.

## strict-ssl

- Default: true
- Type: Boolean

Whether or not to do SSL key validation when making requests to the registry via https.

See also the `ca` config.

## tag

- Default: latest
- Type: String

If you ask npm to install a package and don't tell it a specific version, then it will install the specified tag.

Also the tag that is added to the package@version specified by the `npm tag` command, if no explicit tag is given.

## tag-version-prefix

- Default: "v"
- Type: String

If set, alters the prefix used when tagging a new version when performing a version increment using `npm-version`. To remove the prefix altogether, set it to the empty string: `""`.

Because other tools may rely on the convention that npm version tags look like `v1.0.0`, *only use this property if it is absolutely necessary*. In particular, use care when overriding this setting for public packages.

## tmp

- Default: TMPDIR environment variable, or "/tmp"
- Type: path

Where to store temporary files and folders. All temp files are deleted on success, but left behind on failure for forensic purposes.

## unicode

- Default: false on windows, true on mac/unix systems with a unicode locale

- Type: Boolean

When set to true, npm uses unicode characters in the tree output. When false, it uses ascii characters to draw trees.

### unsafe-perm

- Default: false if running as root, true otherwise
- Type: Boolean

Set to true to suppress the UID/GID switching when running package scripts. If set explicitly to false, then installing as a non-root user will fail.

### usage

- Default: false
- Type: Boolean

Set to show short usage output (like the -H output) instead of complete help when doing [npm-help](#) .

### user

- Default: "nobody"
- Type: String or Number

The UID to set to when running package scripts as root.

### userconfig

- Default: ~/.npmrc
- Type: path

The location of user-level configuration settings.

### umask

- Default: 022
- Type: Octal numeric string in range 0000..0777 (0..511)

The "umask" value to use when setting the file creation mode on files and folders.

Folders and executables are given a mode which is **0777** masked against this value. Other files are given a mode which is **0666** masked against this value. Thus, the defaults are **0755** and **0644** respectively.

### user-agent

- Default: node/{process.version} {process.platform} {process.arch}
- Type: String

Sets a User-Agent to the request header

### version

- Default: false
- Type: boolean

If true, output the npm version and exit successfully.

Only relevant when specified explicitly on the command line.

### versions

- Default: false
- Type: boolean

If true, output the npm version as well as node's **process.versions** map, and exit successfully.

Only relevant when specified explicitly on the command line.

### viewer

- Default: "man" on Posix, "browser" on Windows

- Type: path

The program to use to view help content.

Set to `"browser"` to view html help content in the default web browser.

## See Also

- [npm-config](#)
- [npmrc](#)
- [npm-scripts](#)
- [npm-folders](#)
- [npm](#)

---

Last modified January 21, 2016

Found a typo? Send a [pull request!](#)

## npm-developers

### Description

So, you've decided to use npm to develop (and maybe publish/deploy) your project.

Fantastic!

There are a few things that you need to do above the simple steps that your users will do to install your program.

## About These Documents

These are man pages. If you install npm, you should be able to then do `man npm-thing` to get the documentation on a particular topic, or `npm help thing` to see the same information.

## What is a package

A package is:

- a) a folder containing a program described by a package.json file
- b) a gzipped tarball containing (a)
- c) a url that resolves to (b)
- d) a `<name>@<version>` that is published on the registry with (c)
- e) a `<name>@<tag>` that points to (d)
- f) a `<name>` that has a "latest" tag satisfying (e)
- g) a `git` url that, when cloned, results in (a).

Even if you never publish your package, you can still get a lot of benefits of using npm if you just want to write a node program (a), and perhaps if you also want to be able to easily install it elsewhere after packing it up into a tarball (b).

Git urls can be of the form:

```
git://github.com/user/project.git#commit-ish
git+ssh://user@hostname:project.git#commit-ish
git+http://user@hostname/project/blah.git#commit-ish
git+https://user@hostname/project/blah.git#commit-ish
```

The `commit-ish` can be any tag, sha, or branch which can be supplied as an argument to `git checkout`. The default is `master`.

# The package.json File

You need to have a `package.json` file in the root of your project to do much of anything with npm. That is basically the whole interface.

See [package.json](#) for details about what goes in that file. At the very least, you need:

- **name:** This should be a string that identifies your project. Please do not use the name to specify that it runs on node, or is in JavaScript. You can use the "engines" field to explicitly state the versions of node (or whatever else) that your program requires, and it's pretty well assumed that it's javascript.

It does not necessarily need to match your github repository name.

So, `node-foo` and `bar-js` are bad names. `foo` or `bar` are better.

- **version:** A semver-compatible version.
- **engines:** Specify the versions of node (or whatever else) that your program runs on. The node API changes a lot, and there may be bugs or new functionality that you depend on. Be explicit.
- **author:** Take some credit.
- **scripts:** If you have a special compilation or installation script, then you should put it in the `scripts` object. You should definitely have at least a basic smoke-test command as the "scripts.test" field. See [npm-scripts](#).
- **main:** If you have a single module that serves as the entry point to your program (like what the "foo" package gives you at `require("foo")`), then you need to specify that in the "main" field.
- **directories:** This is an object mapping names to folders. The best ones to include are "lib" and "doc", but if you use "man" to specify a folder full of man pages, they'll get installed just like these ones.

You can use `npm init` in the root of your package in order to get you started with a pretty basic package.json file. See [npm-init](#) for more info.

## Keeping files *out* of your package

Use a `.npmignore` file to keep stuff out of your package. If there's no `.npmignore` file, but there *is* a `.gitignore` file, then npm will ignore the stuff matched by the `.gitignore` file. If you *want* to include something that is excluded by your `.gitignore` file, you can create an empty `.npmignore` file to override it. Like `git`, `npm` looks for `.npmignore` and `.gitignore` files in all subdirectories of your package, not only the root directory.

`.npmignore` files follow the [same pattern rules](#) as `.gitignore` files:

- Blank lines or lines starting with `#` are ignored.
- Standard glob patterns work.
- You can end patterns with a forward slash `/` to specify a directory.
- You can negate a pattern by starting it with an exclamation point `!`.

By default, the following paths and files are ignored, so there's no need to add them to `.npmignore` explicitly:

- `.*.swp`
- `._*`
- `.DS_Store`
- `.git`
- `.hg`
- `.npmrc`
- `.lock-wscript`
- `.svn`
- `.wafpickle-*`
- `config.gypi`
- `CVS`

- `npm-debug.log`

Additionally, everything in `node_modules` is ignored, except for bundled dependencies. npm automatically handles this for you, so don't bother adding `node_modules` to `.npmignore`.

The following paths and files are never ignored, so adding them to `.npmignore` is pointless:

- `package.json`
- `README` (and its variants)
- `CHANGELOG` (and its variants)
- `LICENSE` / `LICENCE`

## Link Packages

`npm link` is designed to install a development package and see the changes in real time without having to keep re-installing it. (You do need to either re-link or `npm rebuild -g` to update compiled packages, of course.)

More info at [npm-link](#).

## Before Publishing: Make Sure Your Package Installs and Works

This is important.

If you can not install it locally, you'll have problems trying to publish it. Or, worse yet, you'll be able to publish it, but you'll be publishing a broken or pointless package. So don't do that.

In the root of your package, do this:

```
npm install . -g
```

That'll show you that it's working. If you'd rather just create a symlink package that points to your working directory, then do this:

```
npm link
```

Use `npm ls -g` to see if it's there.

To test a local install, go into some other folder, and then do:

```
cd ../some-other-folder
npm install ../my-package
```

to install it locally into the `node_modules` folder in that other place.

Then go into the node-repl, and try using `require("my-thing")` to bring in your module's main module.

## Create a User Account

Create a user with the `adduser` command. It works like this:

```
npm adduser
```

and then follow the prompts.

This is documented better in [npm-adduser](#).

## Publish your package

This part's easy. In the root of your folder, do this:



npm publish

You can give publish a url to a tarball, or a filename of a tarball, or a path to a folder.

Note that pretty much **everything in that folder will be exposed** by default. So, if you have secret stuff in there, use a `.npmignore` file to list out the globs to ignore, or publish from a fresh checkout.

## Brag about it

Send emails, write blogs, blab in IRC.

Tell the world how easy it is to install your program!

## See Also

- [npm-faq](#)
- [npm](#)
- [npm-init](#)
- [package.json](#)
- [npm-scripts](#)
- [npm-publish](#)
- [npm-adduser](#)
- [npm-registry](#)

Last modified January 08, 2016

Found a typo? Send a [pull request!](#)

## npm-disputes

## Synopsis

1. Get the author email with `npm owner ls <pkgname>`
2. Email the author, CC [support@npmjs.com](mailto:support@npmjs.com)
3. After a few weeks, if there's no resolution, we'll sort it out.

Don't squat on package names. Publish code or move out of the way.

## Description

There sometimes arise cases where a user publishes a module, and then later, some other user wants to use that name. Here are some common ways that happens (each of these is based on actual events.)

1. Joe writes a JavaScript module `foo`, which is not node-specific. Joe doesn't use node at all. Bob wants to use `foo` in node, so he wraps it in an npm module. Some time later, Joe starts using node, and wants to take over management of his program.
2. Bob writes an npm module `foo`, and publishes it. Perhaps much later, Joe finds a bug in `foo`, and fixes it. He sends a pull request to Bob, but Bob doesn't have the time to deal with it, because he has a new job and a new baby and is focused on his new erlang project, and kind of not involved with node any more. Joe would like to publish a new `foo`, but can't, because the name is taken.
3. Bob writes a 10-line flow-control library, and calls it `foo`, and publishes it to the npm registry. Being a simple little thing, it never really has to be updated. Joe works for Foo Inc, the makers of the critically acclaimed and widely-marketed `foo` JavaScript toolkit framework. They publish it to npm as `foojs`, but people are routinely confused when `npm install foo` is some different thing.
4. Bob writes a parser for the widely-known `foo` file format, because he needs it for work. Then, he gets a new job, and never updates the prototype. Later on, Joe writes a much more complete `foo` parser, but can't publish, because Bob's `foo` is in the way.

The validity of Joe's claim in each situation can be debated. However, Joe's appropriate course of action in each case is the same.

1. `npm owner ls foo`. This will tell Joe the email address of the owner (Bob).
2. Joe emails Bob, explaining the situation **as respectfully as possible**, and what he would like to do with the module name. He adds the npm support staff [support@npmjs.com](mailto:support@npmjs.com) to the CC list of the email. Mention in the email that Bob can run `npm owner add joe foo` to add Joe as an owner of the

`foo` package.

3. After a reasonable amount of time, if Bob has not responded, or if Bob and Joe can't come to any sort of resolution, email support [support@npmjs.com](mailto:support@npmjs.com) and we'll sort it out. ("Reasonable" is usually at least 4 weeks, but extra time is allowed around common holidays.)

## REASONING

In almost every case so far, the parties involved have been able to reach an amicable resolution without any major intervention. Most people really do want to be reasonable, and are probably not even aware that they're in your way.

Module ecosystems are most vibrant and powerful when they are as self-directed as possible. If an admin one day deletes something you had worked on, then that is going to make most people quite upset, regardless of the justification. When humans solve their problems by talking to other humans with respect, everyone has the chance to end up feeling good about the interaction.

## EXCEPTIONS

Some things are not allowed, and will be removed without discussion if they are brought to the attention of the npm registry admins, including but not limited to:

1. Malware (that is, a package designed to exploit or harm the machine on which it is installed).
2. Violations of copyright or licenses (for example, cloning an MIT-licensed program, and then removing or changing the copyright and license statement).
3. Illegal content.
4. "Squatting" on a package name that you *plan* to use, but aren't actually using. Sorry, I don't care how great the name is, or how perfect a fit it is for the thing that someday might happen. If someone wants to use it today, and you're just taking up space with an empty tarball, you're going to be evicted.
5. Putting empty packages in the registry. Packages must have SOME functionality. It can be silly, but it can't be *nothing*. (See also: squatting.)
6. Doing weird things with the registry, like using it as your own personal application database or otherwise putting non-packagey things into it.

If you see bad behavior like this, please report it right away.

## See Also

- [npm-registry](#)
- [npm-owner](#)

---

Last modified January 08, 2016

Found a typo? Send a [pull request!](#)

## npm-orgs

### Description

There are three levels of org users:

1. Super admin, controls billing & adding people to the org.
2. Team admin, manages team membership & package access.
3. Developer, works on packages they are given access to.

The super admin is the only person who can add users to the org because it impacts the monthly bill. The super admin will use the website to manage membership. Every org has a **developers** team that all users are automatically added to.

The team admin is the person who manages team creation, team membership, and package access for teams. The team admin grants package access to teams, not individuals.

The developer will be able to access packages based on the teams they are on. Access is either read-write or read-only.

There are two main commands:

1. `npm team` see [npm-access](#) for more details
2. `npm access` see [npm-team](#) for more details

## Team Admins create teams

- Check who you've added to your org:

```
npm team ls <org>:developers
```

- Each org is automatically given a **developers** team, so you can see the whole list of team members in your org. This team automatically gets read-write access to all packages, but you can change that with the **access** command.

- Create a new team:

```
npm team create <org:team>
```

- Add members to that team:

```
npm team add <org:team> <user>
```

## Publish a package and adjust package access

- In package directory, run

```
npm init --scope=<org>
```

to scope it for your org & publish as usual

- Grant access:

```
npm access grant <read-only|read-write> <org:team> [<package>]
```

- Revoke access:

```
npm access revoke <org:team> [<package>]
```

## Monitor your package access

- See what org packages a team member can access:

```
npm access ls-packages <org> <user>
```

- See packages available to a specific team:

```
npm access ls-packages <org:team>
```

- Check which teams are collaborating on a package:

```
npm access ls-collaborators <pkg>
```

## See Also

- [npm-team](#)
- [npm-access](#)
- [npm-scope](#)

## Description

To resolve packages by name and version, npm talks to a registry website that implements the CommonJS Package Registry specification for reading package info.

Additionally, npm's package registry implementation supports several write APIs as well, to allow for publishing packages and managing user account information.

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

The registry URL used is determined by the scope of the package (see [npm-scope](#)). If no scope is specified, the default registry is used, which is supplied by the `registry` config parameter. See [npm-config](#), [npmrc](#), and [npm-config](#) for more on managing npm's configuration.

## Can I run my own private registry?

Yes!

The easiest way is to replicate the couch database, and use the same (or similar) design doc to implement the APIs.

If you set up continuous replication from the official CouchDB, and then set your internal CouchDB as the registry config, then you'll be able to read any published packages, in addition to your private ones, and by default will only publish internally.

If you then want to publish a package for the whole world to see, you can simply override the `--registry` option for that `publish` command.

## I don't want my package published in the official registry. It's private.

Set `"private": true` in your package.json to prevent it from being published at all, or `"publishConfig":{"registry":"http://my-internal-registry.local"}` to force it to be published only to your internal registry.

See [package.json](#) for more info on what goes in the package.json file.

## Will you replicate from my registry into the public one?

No. If you want things to be public, then publish them into the public registry using npm. What little security there is would be for nought otherwise.

## Do I have to use couchdb to build a registry that npm can talk to?

No, but it's way easier. Basically, yes, you do, or you have to effectively implement the entire CouchDB API anyway.

## Is there a website or something to see package docs and such?

Yes, head over to <https://npmjs.com/>

## See Also

- [npm-config](#)
- [npm-config](#)
- [npmrc](#)
- [npm-developers](#)
- [npm-disputes](#)

## npm-removal

# Synopsis

So sad to see you go.

```
sudo npm uninstall npm -g
```

Or, if that fails, get the npm source code, and do:

```
sudo make uninstall
```

## More Severe Uninstalling

Usually, the above instructions are sufficient. That will remove npm, but leave behind anything you've installed.

If that doesn't work, or if you require more drastic measures, continue reading.

Note that this is only necessary for globally-installed packages. Local installs are completely contained within a project's `node_modules` folder. Delete that folder, and everything is gone (unless a package's install script is particularly ill-behaved).

This assumes that you installed node and npm in the default place. If you configured node with a different `--prefix`, or installed npm with a different prefix setting, then adjust the paths accordingly, replacing `/usr/local` with your install prefix.

To remove everything npm-related manually:

```
rm -rf /usr/local/{lib/node{,/.npm,_modules},bin,share/man}/npm*
```

If you installed things *with* npm, then your best bet is to uninstall them with npm first, and then install them again once you have a proper install. This can help find any symlinks that are lying around:

```
ls -laF /usr/local/{lib/node{,/.npm},bin,share/man} | grep npm
```

Prior to version 0.3, npm used shim files for executables and node modules. To track those down, you can do the following:

```
find /usr/local/{lib/node,bin} -exec grep -l npm {} \; ;
```

(This is also in the README file.)

## See Also

- README
- [npm-uninstall](#)
- [npm-prune](#)

## npm-scope

# Description

All npm packages have a name. Some package names also have a scope. A scope follows the usual rules for package names (url-safe characters, no leading dots or underscores). When used in package names, preceded by an @-symbol and followed by a slash, e.g.

```
@somescope/somepackagename
```

Scopes are a way of grouping related packages together, and also affect a few things about the way npm treats the package.

Scoped packages are supported by the public npm registry. The npm client is backwards-compatible with un-scoped registries, so it can be used to work with scoped and un-scoped registries at the same time.

## Installing scoped packages

Scoped packages are installed to a sub-folder of the regular installation folder, e.g. if your other packages are installed in `node_modules/packagename`, scoped modules will be in `node_modules/@myorg/packagename`. The scope folder (`@myorg`) is simply the name of the scope preceded by an @-symbol, and can contain any number of scoped packages.

A scoped package is installed by referencing it by name, preceded by an @-symbol, in `npm install`:

```
npm install @myorg/mypackage
```

Or in `package.json`:

```
"dependencies": {  
  "@myorg/mypackage": "^1.3.0"  
}
```

Note that if the @-symbol is omitted in either case npm will instead attempt to install from GitHub; see [npm-install](#).

## Requiring scoped packages

Because scoped packages are installed into a scope folder, you have to include the name of the scope when requiring them in your code, e.g.

```
require('@myorg/mypackage')
```

There is nothing special about the way Node treats scope folders, this is just specifying to require the module `mypackage` in the folder called `@myorg`.

## Publishing scoped packages

Scoped packages can be published to any registry that supports them, including the public npm registry.

(As of 2015-04-19, the public npm registry **does** support scoped packages)

If you wish, you may associate a scope with a registry; see below.

### Publishing public scoped packages to the public npm registry

To publish a public scoped package, you must specify `--access public` with the initial publication. This will publish the package and set access to `public` as if you had run `npm access public` after publishing.

### Publishing private scoped packages to the npm registry

To publish a private scoped package to the npm registry, you must have an [npm Private Modules](#) account.

You can then publish the module with `npm publish` or `npm publish --access restricted`, and it will be present in the npm registry, with restricted access. You can then change the access permissions, if desired, with `npm access` or on the [npmjs.com](#) website.

# Associating a scope with a registry

Scopes can be associated with a separate registry. This allows you to seamlessly use a mix of packages from the public npm registry and one or more private registries, such as npm Enterprise.

You can associate a scope with a registry at login, e.g.

```
npm login --registry=http://reg.example.com --scope=@myco
```

Scopes have a many-to-one relationship with registries: one registry can host multiple scopes, but a scope only ever points to one registry.

You can also associate a scope with a registry using `npm config` :

```
npm config set @myco:registry http://reg.example.com
```

Once a scope is associated with a registry, any `npm install` for a package with that scope will request packages from that registry instead. Any `npm publish` for a package name that contains the scope will be published to that registry instead.

## See Also

- [npm-install](#)
- [npm-publish](#)
- [npm-access](#)

---

Last modified January 08, 2016

Found a typo? Send a [pull request!](#)

## npm-scripts

## Description

npm supports the "scripts" property of the package.json script, for the following scripts:

- prepublish: Run BEFORE the package is published. (Also run on local `npm install` without any arguments.)
- publish, postpublish: Run AFTER the package is published.
- preinstall: Run BEFORE the package is installed
- install, postinstall: Run AFTER the package is installed.
- preuninstall, uninstall: Run BEFORE the package is uninstalled.
- postuninstall: Run AFTER the package is uninstalled.
- preversion, version: Run BEFORE bump the package version.
- postversion: Run AFTER bump the package version.
- pretest, test, posttest: Run by the `npm test` command.
- prestop, stop, poststop: Run by the `npm stop` command.
- prestart, start, poststart: Run by the `npm start` command.
- prerestart, restart, postrestart: Run by the `npm restart` command. Note: `npm restart` will run the stop and start scripts if no `restart` script is provided.

Additionally, arbitrary scripts can be executed by running `npm run-script <pkg> <stage>`. *Pre* and *post* commands with matching names will be run for those as well (e.g. `premyscript`, `myscript`, `postmyscript`).

## COMMON USES

If you need to perform operations on your package before it is used, in a way that is not dependent on the operating system or architecture of the target system, use a `prepublish` script. This includes tasks such as:

- Compiling CoffeeScript source code into JavaScript.
- Creating minified versions of JavaScript source code.
- Fetching remote resources that your package will use.

The advantage of doing these things at **prepublish** time is that they can be done once, in a single place, thus reducing complexity and variability. Additionally, this means that:

- You can depend on **coffee-script** as a **devDependency**, and thus your users don't need to have it installed.
- You don't need to include minifiers in your package, reducing the size for your users.
- You don't need to rely on your users having **curl** or **wget** or other system tools on the target machines.

## DEFAULT VALUES

npm will default some script values based on package contents.

- **"start": "node server.js"** :

If there is a **server.js** file in the root of your package, then npm will default the **start** command to **node server.js**.

- **"install": "node-gyp rebuild"** :

If there is a **binding.gyp** file in the root of your package, npm will default the **install** command to compile using node-gyp.

## USER

If npm was invoked with root privileges, then it will change the uid to the user account or uid specified by the **user** config, which defaults to **nobody**. Set the **unsafe-perm** flag to run scripts with root privileges.

## ENVIRONMENT

Package scripts run in an environment where many pieces of information are made available regarding the setup of npm and the current state of the process.

### path

If you depend on modules that define executable scripts, like test suites, then those executables will be added to the **PATH** for executing the scripts. So, if your **package.json** has this:

```
{ "name" : "foo"
, "dependencies" : { "bar" : "0.1.x" }
, "scripts": { "start" : "bar ./test" } }
```

then you could run **npm start** to execute the **bar** script, which is exported into the **node\_modules/.bin** directory on **npm install**.

### package.json vars

The **package.json** fields are tacked onto the **npm\_package\_** prefix. So, for instance, if you had **{ "name": "foo", "version": "1.2.5" }** in your **package.json** file, then your package scripts would have the **npm\_package\_name** environment variable set to "foo", and the **npm\_package\_version** set to "1.2.5".

### configuration

Configuration parameters are put in the environment with the **npm\_config\_** prefix. For instance, you can view the effective **root** config by checking the **npm\_config\_root** environment variable.

### Special: package.json "config" object

The **package.json** "config" keys are overwritten in the environment if there is a config param of **<name>[@<version>]:<key>**. For example, if the **package.json** has this:



```
{ "name" : "foo"
  , "config" : { "port" : "8080" }
  , "scripts" : { "start" : "node server.js" } }
```

and the server.js is this:

```
http.createServer(...).listen(process.env.npm_package_config_port)
```

then the user could change the behavior by doing:

```
npm config set foo:port 80
```

### current lifecycle event

Lastly, the `npm_lifecycle_event` environment variable is set to whichever stage of the cycle is being executed. So, you could have a single script used for different parts of the process which switches based on what's currently happening.

Objects are flattened following this format, so if you had `{"scripts":{"install":"foo.js"}}` in your package.json, then you'd see this in the script:

```
process.env.npm_package_scripts_install === "foo.js"
```

## EXAMPLES

For example, if your package.json contains this:

```
{ "scripts" :
  { "install" : "scripts/install.js"
    , "postinstall" : "scripts/install.js"
    , "uninstall" : "scripts/uninstall.js"
  }
}
```

then `scripts/install.js` will be called for the install and post-install stages of the lifecycle, and `scripts/uninstall.js` will be called when the package is uninstalled. Since `scripts/install.js` is running for two different phases, it would be wise in this case to look at the `npm_lifecycle_event` environment variable.

If you want to run a make command, you can do so. This works just fine:

```
{ "scripts" :
  { "preinstall" : "./configure"
    , "install" : "make && make install"
    , "test" : "make test"
  }
}
```

## EXITING

Scripts are run by passing the line as a script argument to `sh`.

If the script exits with a code other than 0, then this will abort the process.

Note that these script files don't have to be nodejs or even javascript programs. They just have to be some kind of executable file.

## HOOK SCRIPTS

If you want to run a specific script at a specific lifecycle event for ALL packages, then you can use a hook script.

Place an executable file at `node_modules/.hooks/{eventname}`, and it'll get run for all packages when they are going through that point in the package lifecycle for any packages installed in that root.

Hook scripts are run exactly the same way as package.json scripts. That is, they are in a separate child process, with the env described above.

# BEST PRACTICES

- Don't exit with a non-zero error code unless you *really* mean it. Except for uninstall scripts, this will cause the npm action to fail, and potentially be rolled back. If the failure is minor or only will prevent some optional features, then it's better to just print a warning and exit successfully.
- Try not to use scripts to do what npm can do for you. Read through [package.json](#) to see all the things that you can specify and enable by simply describing your package appropriately. In general, this will lead to a more robust and consistent state.
- Inspect the env to determine where to put things. For instance, if the `npm_config_binroot` environment variable is set to `/home/user/bin`, then don't try to install executables into `/usr/local/bin`. The user probably set it up that way for a reason.
- Don't prefix your script commands with "sudo". If root permissions are required for some reason, then it'll fail with that error, and the user will sudo the npm command in question.
- Don't use `install`. Use a `.gyp` file for compilation, and `prepublish` for anything else. You should almost never have to explicitly set a preinstall or install script. If you are doing this, please consider if there is another option. The only valid use of `install` or `preinstall` scripts is for compilation which must be done on the target architecture.

## See Also

- [npm-run-script](#)
- [package.json](#)
- [npm-developers](#)
- [npm-install](#)

Last modified January 08, 2016

Found a typo? Send a [pull request!](#)

## semver

## Usage

```
$ npm install semver

semver.valid('1.2.3') // '1.2.3'
semver.valid('a.b.c') // null
semver.clean('v1.2.3') // '1.2.3'
semver.satisfies('1.2.3', '1.x || >=2.5.0 || 5.0.0 - 7.2.3') // true
semver.gt('1.2.3', '9.8.7') // false
semver.lt('1.2.3', '9.8.7') // true
```

As a command-line utility:

```
$ semver -h

Usage: semver <version> [<version> [...]] [-r <range> | -i <inc> | --preid <identifier> | -l | -rv]
Test if version(s) satisfy the supplied range(s), and sort them.

Multiple versions or ranges may be supplied, unless increment
option is specified. In that case, only a single version may
be used, and it is incremented by the specified level

Program exits successfully if any valid version satisfies
all supplied ranges, and prints all satisfying versions.

If no versions are valid, or ranges are not satisfied,
then exits failure.

Versions are printed in ascending order, so supplying
multiple versions to the utility will just sort them.
```

## Versions

A "version" is described by the **v2.0.0** specification found at <http://semver.org/>.

A leading "=" or "v" character is stripped off and ignored.

## Ranges

A **version range** is a set of **comparators** which specify versions that satisfy the range.

A **comparator** is composed of an **operator** and a **version**. The set of primitive **operators** is:

- < Less than
- <= Less than or equal to
- > Greater than
- >= Greater than or equal to
- = Equal. If no operator is specified, then equality is assumed, so this operator is optional, but MAY be included.

For example, the comparator **>=1.2.7** would match the versions **1.2.7**, **1.2.8**, **2.5.3**, and **1.3.9**, but not the versions **1.2.6** or **1.1.0**.

Comparators can be joined by whitespace to form a **comparator set**, which is satisfied by the **intersection** of all of the comparators it includes.

A range is composed of one or more comparator sets, joined by **||**. A version matches a range if and only if every comparator in at least one of the **||**-separated comparator sets is satisfied by the version.

For example, the range **>=1.2.7 <1.3.0** would match the versions **1.2.7**, **1.2.8**, and **1.2.99**, but not the versions **1.2.6**, **1.3.0**, or **1.1.0**.

The range **1.2.7 || >=1.2.9 <2.0.0** would match the versions **1.2.7**, **1.2.9**, and **1.4.6**, but not the versions **1.2.8** or **2.0.0**.

### Prerelease Tags

If a version has a prerelease tag (for example, **1.2.3-alpha.3**) then it will only be allowed to satisfy comparator sets if at least one comparator with the same **[major, minor, patch]** tuple also has a prerelease tag.

For example, the range **>1.2.3-alpha.3** would be allowed to match the version **1.2.3-alpha.7**, but it would *not* be satisfied by **3.4.5-alpha.9**, even though **3.4.5-alpha.9** is technically "greater than" **1.2.3-alpha.3** according to the SemVer sort rules. The version range only accepts prerelease tags on the **1.2.3** version. The version **3.4.5** *would* satisfy the range, because it does not have a prerelease flag, and **3.4.5** is greater than **1.2.3-alpha.7**.

The purpose for this behavior is twofold. First, prerelease versions frequently are updated very quickly, and contain many breaking changes that are (by the author's design) not yet fit for public consumption. Therefore, by default, they are excluded from range matching semantics.

Second, a user who has opted into using a prerelease version has clearly indicated the intent to use *that specific* set of alpha/beta/rc versions. By including a prerelease tag in the range, the user is indicating that they are aware of the risk. However, it is still not appropriate to assume that they have opted into taking a similar risk on the *next* set of prerelease versions.

### Prerelease Identifiers

The method **.inc** takes an additional **identifier** string argument that will append the value of the string as a prerelease identifier:

```
> semver.inc('1.2.3', 'prerelease', 'beta')
'1.2.4-beta.0'
```

command-line example:

```
$ semver 1.2.3 -i prerelease --preid beta
1.2.4-beta.0
```

Which then can be used to increment further:

```
$ semver 1.2.4-beta.0 -i prerelease
1.2.4-beta.1
```

### Advanced Range Syntax

Advanced range syntax desugars to primitive comparators in deterministic ways.

Advanced ranges may be combined in the same way as primitive comparators using white space or **||**.

### Hyphen Ranges **X.Y.Z - A.B.C**

Specifies an inclusive set.

- `1.2.3 - 2.3.4 := >=1.2.3 <=2.3.4`

If a partial version is provided as the first version in the inclusive range, then the missing pieces are replaced with zeroes.

- `1.2 - 2.3.4 := >=1.2.0 <=2.3.4`

If a partial version is provided as the second version in the inclusive range, then all versions that start with the supplied parts of the tuple are accepted, but nothing that would be greater than the provided tuple parts.

- `1.2.3 - 2.3 := >=1.2.3 <2.4.0`
- `1.2.3 - 2 := >=1.2.3 <3.0.0`

#### X-Ranges `1.2.x` `1.X` `1.2.*` `*`

Any of `X`, `x`, or `*` may be used to "stand in" for one of the numeric values in the `[major, minor, patch]` tuple.

- `*` `:= >=0.0.0` (Any version satisfies)
- `1.x` `:= >=1.0.0 <2.0.0` (Matching major version)
- `1.2.x` `:= >=1.2.0 <1.3.0` (Matching major and minor versions)

A partial version range is treated as an X-Range, so the special character is in fact optional.

- `""` (empty string) `:= *` `:= >=0.0.0`
- `1` `:= 1.x.x` `:= >=1.0.0 <2.0.0`
- `1.2` `:= 1.2.x` `:= >=1.2.0 <1.3.0`

#### Tilde Ranges `~1.2.3` `~1.2` `~1`

Allows patch-level changes if a minor version is specified on the comparator. Allows minor-level changes if not.

- `~1.2.3` `:= >=1.2.3 <1.(2+1).0` `:= >=1.2.3 <1.3.0`
- `~1.2` `:= >=1.2.0 <1.(2+1).0` `:= >=1.2.0 <1.3.0` (Same as `1.2.x`)
- `~1` `:= >=1.0.0 <(1+1).0.0` `:= >=1.0.0 <2.0.0` (Same as `1.x`)
- `~0.2.3` `:= >=0.2.3 <0.(2+1).0` `:= >=0.2.3 <0.3.0`
- `~0.2` `:= >=0.2.0 <0.(2+1).0` `:= >=0.2.0 <0.3.0` (Same as `0.2.x`)
- `~0` `:= >=0.0.0 <(0+1).0.0` `:= >=0.0.0 <1.0.0` (Same as `0.x`)
- `~1.2.3-beta.2` `:= >=1.2.3-beta.2 <1.3.0` Note that prereleases in the `1.2.3` version will be allowed, if they are greater than or equal to `beta.2`. So, `1.2.3-beta.4` would be allowed, but `1.2.4-beta.2` would not, because it is a prerelease of a different `[major, minor, patch]` tuple.

#### Caret Ranges `^1.2.3` `^0.2.5` `^0.0.4`

Allows changes that do not modify the left-most non-zero digit in the `[major, minor, patch]` tuple. In other words, this allows patch and minor updates for versions `1.0.0` and above, patch updates for versions `0.X >=0.1.0`, and *no* updates for versions `0.0.X`.

Many authors treat a `0.x` version as if the `x` were the major "breaking-change" indicator.

Caret ranges are ideal when an author may make breaking changes between `0.2.4` and `0.3.0` releases, which is a common practice. However, it presumes that there will *not* be breaking changes between `0.2.4` and `0.2.5`. It allows for changes that are presumed to be additive (but non-breaking), according to commonly observed practices.

- `^1.2.3` `:= >=1.2.3 <2.0.0`
- `^0.2.3` `:= >=0.2.3 <0.3.0`
- `^0.0.3` `:= >=0.0.3 <0.0.4`
- `^1.2.3-beta.2` `:= >=1.2.3-beta.2 <2.0.0` Note that prereleases in the `1.2.3` version will be allowed, if they are greater than or equal to `beta.2`. So, `1.2.3-beta.4` would be allowed, but `1.2.4-beta.2` would not, because it is a prerelease of a different `[major, minor, patch]` tuple.
- `^0.0.3-beta` `:= >=0.0.3-beta <0.0.4` Note that prereleases in the `0.0.3` version *only* will be allowed, if they are greater than or equal to `beta`. So, `0.0.3-pr.2` would be allowed.

When parsing caret ranges, a missing `patch` value desugars to the number `0`, but will allow flexibility within that value, even if the major and minor versions are both `0`.

- `^1.2.x` `:= >=1.2.0 <2.0.0`
- `^0.0.x` `:= >=0.0.0 <0.1.0`
- `^0.0` `:= >=0.0.0 <0.1.0`

A missing **minor** and **patch** values will desugar to zero, but also allow flexibility within those values, even if the major version is zero.

- `^1.x := >=1.0.0 <2.0.0`
- `^0.x := >=0.0.0 <1.0.0`

## Range Grammar

Putting all this together, here is a Backus-Naur grammar for ranges, for the benefit of parser authors:

```
range-set ::= range ( logical-or range ) *
logical-or ::= ( ' ' ) * '|' ( ' ' ) *
range ::= hyphen | simple ( ' ' simple ) * | ''
hyphen ::= partial ' - ' partial
simple ::= primitive | partial | tilde | caret
primitive ::= ( '<' | '>' | '>=' | '<=' | '=' | ) partial
partial ::= xr ( '.' xr ( '.' xr qualifier ? )? )?
xr ::= 'x' | 'X' | '*' | nr
nr ::= '0' | ['1'-'9']['0'-'9']+
tilde ::= '~' partial
caret ::= '^' partial
qualifier ::= ( '-' pre )? ( '+' build )?
pre ::= parts
build ::= parts
parts ::= part ( '.' part ) *
part ::= nr | [-0-9A-Za-z]+
```

# Functions

All methods and classes take a final **loose** boolean argument that, if true, will be more forgiving about not-quite-valid semver strings. The resulting output will always be 100% strict, of course.

Strict-mode Comparators and Ranges will be strict about the SemVer strings that they parse.

- **valid(v)** : Return the parsed version, or null if it's not valid.
- **inc(v, release)** : Return the version incremented by the release type ( **major** , **premajor** , **minor** , **preminor** , **patch** , **prepatch** , or **prerelease** ), or null if it's not valid
  - **premajor** in one call will bump the version up to the next major version and down to a prerelease of that major version. **preminor** , and **prepatch** work the same way.
  - If called from a non-prerelease version, the **prerelease** will work the same as **prepatch** . It increments the patch version, then makes a prerelease. If the input version is already a prerelease it simply increments it.
- **major(v)** : Return the major version number.
- **minor(v)** : Return the minor version number.
- **patch(v)** : Return the patch version number.

## Comparison

- **gt(v1, v2)** : **v1 > v2**
- **gte(v1, v2)** : **v1 >= v2**
- **lt(v1, v2)** : **v1 < v2**
- **lte(v1, v2)** : **v1 <= v2**
- **eq(v1, v2)** : **v1 == v2** This is true if they're logically equivalent, even if they're not the exact same string. You already know how to compare strings.
- **neq(v1, v2)** : **v1 != v2** The opposite of **eq** .
- **cmp(v1, comparator, v2)** : Pass in a comparison string, and it'll call the corresponding function above. "===" and "!=" do simple string comparison, but are included for completeness. Throws if an invalid comparison string is provided.
- **compare(v1, v2)** : Return **0** if **v1 == v2** , or **1** if **v1** is greater, or **-1** if **v2** is greater. Sorts in ascending order if passed to **Array.sort()** .
- **rcompare(v1, v2)** : The reverse of compare. Sorts an array of versions in descending order when passed to **Array.sort()** .
- **diff(v1, v2)** : Returns difference between two versions by the release type ( **major** , **premajor** , **minor** , **preminor** , **patch** , **prepatch** , or **prerelease** ), or null if the versions are the same.

## Ranges

- **validRange(range)** : Return the valid range or null if it's not valid
- **satisfies(version, range)** : Return true if the version satisfies the range.
- **maxSatisfying(versions, range)** : Return the highest version in the list that satisfies the range, or **null** if none of them do.
- **gtr(version, range)** : Return **true** if version is greater than all the versions possible in the range.
- **ltr(version, range)** : Return **true** if version is less than all the versions possible in the range.

- **outside(version, range, hilo)** : Return true if the version is outside the bounds of the range in either the high or low direction. The **hilo** argument must be either the string **'>'** or **'<'**. (This is the function called by **gtr** and **ltr**.)

Note that, since ranges may be non-contiguous, a version might not be greater than a range, less than a range, or satisfy a range! For example, the range **1.2 <1.2.9 || >2.0.0** would have a hole from **1.2.9** until **2.0.0**, so the version **1.2.10** would not be greater than the range (because **2.0.1** satisfies, which is higher), nor less than the range (since **1.2.8** satisfies, which is lower), and it also does not satisfy the range.

If you want to know if a version satisfies or does not satisfy a range, use the **satisfies(version, range)** function.

---

Last modified January 08, 2016

Found a typo? Send a [pull request!](#)

## npm-access

# Synopsis

```
npm access public [<package>]
npm access restricted [<package>]

npm access grant <read-only|read-write> <scope:team> [<package>]
npm access revoke <scope:team> [<package>]

npm access ls-packages [<user>|<scope>|<scope:team>]
npm access ls-collaborators [<package> [<user>]]
npm access edit [<package>]
```

## Description

Used to set access controls on private packages.

For all of the subcommands, **npm access** will perform actions on the packages in the current working directory if no package name is passed to the subcommand.

- **public / restricted**: Set a package to be either publicly accessible or restricted.
- **grant / revoke**: Add or remove the ability of users and teams to have read-only or read-write access to a package.
- **ls-packages**:  
Show all of the packages a user or a team is able to access, along with the access level, except for read-only public packages (it won't print the whole registry listing)
- **ls-collaborators**: Show all of the access privileges for a package. Will only show permissions for packages to which you have at least read access. If **<user>** is passed in, the list is filtered only to teams *that* user happens to belong to.
- **edit**: Set the access privileges for a package at once using **\$EDITOR**.

## DETAILS

**npm access** always operates directly on the current registry, configurable from the command line using **--registry=<registry url>**.

Unscoped packages are *always public*.

Scoped packages *default to restricted*, but you can either publish them as public using **npm publish --access=public**, or set their access as public using **npm access public** after the initial publish.

You must have privileges to set the access of a package:

- You are an owner of an unscoped or scoped package.
- You are a member of the team that owns a scope.
- You have been given read-write privileges for a package, either as a member of a team or directly as an owner.

If your account is not paid, then attempts to publish scoped packages will fail with an HTTP 402 status code (logically enough), unless you use `--access=public`.

Management of teams and team memberships is done with the `npm team` command.

## See Also

- [npm-team](#)
- [npm-publish](#)
- [npm-config](#)
- [npm-registry](#)

---

Last modified January 08, 2016

Found a typo? Send a [pull request!](#)

## npm-adduser

## Synopsis

```
npm adduser [--registry=url] [--scope=@orgname] [--always-auth]
```

## Description

Create or verify a user named `<username>` in the specified registry, and save the credentials to the `.npmrc` file. If no registry is specified, the default registry will be used (see [npm-config](#)).

The username, password, and email are read in from prompts.

To reset your password, go to <https://www.npmjs.com/forgot>

To change your email address, go to <https://www.npmjs.com/email-edit>

You may use this command multiple times with the same user account to authorize on a new machine. When authenticating on a new machine, the username, password and email address must all match with your existing record.

`npm login` is an alias to `adduser` and behaves exactly the same way.

## Configuration

### registry

Default: <https://registry.npmjs.org/>

The base URL of the npm package registry. If `scope` is also specified, this registry will only be used for packages with that scope. See [npm-scope](#).

### scope

Default: none

If specified, the user and login credentials given will be associated with the specified scope. See [npm-scope](#). You can use both at the same time, e.g.

```
npm adduser --registry=http://myregistry.example.com --scope=@myco
```

This will set a registry for the given scope and login or create a user for that registry at the same time.

### always-auth

Default: false

If specified, save configuration indicating that all requests to the given registry should include authorization information. Useful for private registries. Can be used with `--registry` and / or `--scope`, e.g.

```
npm adduser --registry=http://private-registry.example.com --always-auth
```

This will ensure that all requests to that registry (including for tarballs) include an authorization header. See `always-auth` in [npm-config](#) for more details on `always-auth`. Registry-specific configuration of `always-auth` takes precedence over any global configuration.

## See Also

- [npm-registry](#)
- [npm-config](#)
- [npm-config](#)
- [npmrc](#)
- [npm-owner](#)
- [npm-whoami](#)

---

Last modified January 08, 2016

Found a typo? Send a [pull request!](#)

## npm-bin

## Synopsis

```
npm bin [-g|--global]
```

## Description

Print the folder where npm will install executables.

## See Also

- [npm-prefix](#)
- [npm-root](#)
- [npm-folders](#)
- [npm-config](#)
- [npm-config](#)
- [npmrc](#)

---

Last modified January 08, 2016

Found a typo? Send a [pull request!](#)



## npm-bugs

# Synopsis

```
npm bugs [<pkgname>]
```

## Description

This command tries to guess at the likely location of a package's bug tracker URL, and then tries to open it using the `--browser` config param. If no package name is provided, it will search for a `package.json` in the current folder and use the `name` property.

## Configuration

### browser

- Default: OS X: `"open"` , Windows: `"start"` , Others: `"xdg-open"`
- Type: String

The browser that is called by the `npm bugs` command to open websites.

### registry

- Default: <https://registry.npmjs.org/>
- Type: url

The base URL of the npm package registry.

## See Also

- [npm-docs](#)
- [npm-view](#)
- [npm-publish](#)
- [npm-registry](#)
- [npm-config](#)
- [npm-config](#)
- [npmrc](#)
- [package.json](#)

---

Last modified January 08, 2016

Found a typo? Send a [pull request!](#)

## npm-build

# Synopsis

```
npm build [<package-folder>]
```

- `<package-folder>` : A folder containing a `package.json` file in its root.

# Description

This is the plumbing command called by `npm link` and `npm install`.

It should generally be called during installation, but if you need to run it directly, run:

```
npm run-script build
```

## See Also

- [npm-install](#)
- [npm-link](#)
- [npm-scripts](#)
- [package.json](#)

---

Last modified January 08, 2016

Found a typo? Send a [pull request!](#)

## npm-bundle

# Description

The `npm bundle` command has been removed in 1.0, for the simple reason that it is no longer necessary, as the default behavior is now to install packages into the local space.

Just use `npm install` now to do what `npm bundle` used to do.

## See Also

- [npm-install](#)

---

Last modified January 08, 2016

Found a typo? Send a [pull request!](#)

## npm-cache

# Synopsis

```
npm cache add <tarball file>
npm cache add <folder>
npm cache add <tarball url>
npm cache add <name>@<version>

npm cache ls [<path>]

npm cache clean [<path>]
```

# Description

Used to add, list, or clear the npm cache folder.

- `add`: Add the specified package to the local cache. This command is primarily intended to be used internally by npm, but it can provide a way to add data to the local installation cache explicitly.
- `ls`: Show the data in the cache. Argument is a path to show in the cache folder. Works a bit like the `find` program, but limited by the `depth` config.
- `clean`: Delete data out of the cache folder. If an argument is provided, then it specifies a subpath to delete. If no argument is provided, then the entire cache is cleared.

## DETAILS

npm stores cache data in the directory specified in `npm config get cache`. For each package that is added to the cache, three pieces of information are stored in `{cache}/{name}/{version}`:

- `.../package/package.json`: The package.json file, as npm sees it.
- `.../package.tgz`: The tarball for that version.

Additionally, whenever a registry request is made, a `.cache.json` file is placed at the corresponding URI, to store the ETag and the requested data. This is stored in `{cache}/{hostname}/{path}/.cache.json`.

Commands that make non-essential registry requests (such as `search` and `view`, or the completion scripts) generally specify a minimum timeout. If the `.cache.json` file is younger than the specified timeout, then they do not make an HTTP request to the registry.

## Configuration

### cache

Default: `~/.npm` on Posix, or `%AppData%/npm-cache` on Windows.

The root cache folder.

## See Also

- [npm-folders](#)
- [npm-config](#)
- [npm-config](#)
- [npmrc](#)
- [npm-install](#)
- [npm-publish](#)
- [npm-pack](#)

---

Last modified January 08, 2016

Found a typo? Send a [pull request!](#)

## npm-completion

## Synopsis

```
source <(npm completion)
```

## Description

Enables tab-completion in all npm commands.

The synopsis above loads the completions into your current shell. Adding it to your `~/.bashrc` or `~/.zshrc` will make the completions available everywhere:

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

You may of course also pipe the output of `npm completion` to a file such as `/usr/local/etc/bash_completion.d/npm` if you have a system that will read that file for you.

When `COMP_CWORD`, `COMP_LINE`, and `COMP_POINT` are defined in the environment, `npm completion` acts in "plumbing mode", and outputs completions based on the arguments.

## See Also

- [npm-developers](#)
- [npm-faq](#)
- [npm](#)

---

Last modified January 08, 2016

Found a typo? Send a [pull request!](#)

## npm-config

## Synopsis

```
npm config set <key> <value> [-g|--global]
npm config get <key>
npm config delete <key>
npm config list
npm config edit
npm get <key>
npm set <key> <value> [-g|--global]
```

## Description

npm gets its config settings from the command line, environment variables, `npmrc` files, and in some cases, the `package.json` file.

See [npmrc](#) for more information about the npmrc files.

See [npm-config](#) for a more thorough discussion of the mechanisms involved.

The `npm config` command can be used to update and edit the contents of the user and global npmrc files.

## Sub-commands

Config supports the following sub-commands:

### set

```
npm config set key value
```

Sets the config key to the value.

If value is omitted, then it sets it to "true".

### get

```
npm config get key
```

Echo the config value to stdout.

### list

```
npm config list
```

Show all the config settings.

### delete

```
npm config delete key
```

Deletes the key from all configuration files.

### edit

```
npm config edit
```

Opens the config file in an editor. Use the `--global` flag to edit the global config.

## See Also

- [npm-folders](#)
- [npm-config](#)
- [package.json](#)
- [npmrc](#)
- [npm](#)

---

Last modified January 08, 2016

Found a typo? Send a [pull request!](#)

## npm-dedupe

## Synopsis

```
npm dedupe
npm ddp
```

## Description

Searches the local package tree and attempts to simplify the overall structure by moving dependencies further up the tree, where they can be more effectively shared by multiple dependent packages.

For example, consider this dependency graph:

```
a
+-- b <-- depends on c@1.0.x
|   `-- c@1.0.3
+-- d <-- depends on c@~1.0.9
|   `-- c@1.0.10
```

In this case, [npm-dedupe](#) will transform the tree to:

```
a
+-- b
+-- d
`-- c@1.0.10
```

Because of the hierarchical nature of node's module lookup, b and d will both get their dependency met by the single c package at the root level of the tree.

The deduplication algorithm walks the tree, moving each dependency as far up in the tree as possible, even if duplicates are not found. This will result in both a flat and deduplicated tree.

If a suitable version exists at the target location in the tree already, then it will be left untouched, but the other duplicates will be deleted.

Arguments are ignored. Dedupe always acts on the entire tree.

Modules

Note that this operation transforms the dependency tree, but will never result in new modules being installed.

## See Also

- [npm-ls](#)
- [npm-update](#)
- [npm-install](#)

---

Last modified January 08, 2016

Found a typo? Send a [pull request!](#)

## npm-deprecate

## Synopsis

```
npm deprecate <pkg>[@<version>] <message>
```

## Description

This command will update the npm registry entry for a package, providing a deprecation warning to all who attempt to install it.

It works on version ranges as well as specific versions, so you can do something like this:

```
npm deprecate my-thing@"< 0.2.3" "critical bug fixed in v0.2.3"
```

Note that you must be the package owner to deprecate something. See the **owner** and **adduser** help topics.

To un-deprecate a package, specify an empty string ( `""` ) for the **message** argument.

## See Also

- [npm-publish](#)
- [npm-registry](#)

---

Last modified January 08, 2016

Found a typo? Send a [pull request!](#)

## npm-dist-tag

# Synopsis

```
npm dist-tag add <pkg>@<version> [<tag>]
npm dist-tag rm <pkg> <tag>
npm dist-tag ls [<pkg>]
```

# Description

Add, remove, and enumerate distribution tags on a package:

- **add**: Tags the specified version of the package with the specified tag, or the `--tag` config if not specified.
- **rm**: Clear a tag that is no longer in use from the package.
- **ls**: Show all of the dist-tags for a package, defaulting to the package in the current prefix.

A tag can be used when installing packages as a reference to a version instead of using a specific version number:

```
npm install <name>@<tag>
```

When installing dependencies, a preferred tagged version may be specified:

```
npm install --tag <tag>
```

This also applies to `npm dedupe`.

Publishing a package sets the `latest` tag to the published version unless the `--tag` option is used. For example, `npm publish --tag=beta`.

By default, `npm install <pkg>` (without any `@<version>` or `@<tag>` specifier) installs the `latest` tag.

# PURPOSE

Tags can be used to provide an alias instead of version numbers.

For example, a project might choose to have multiple streams of development and use a different tag for each stream, e.g., `stable`, `beta`, `dev`, `canary`.

By default, the `latest` tag is used by npm to identify the current version of a package, and `npm install <pkg>` (without any `@<version>` or `@<tag>` specifier) installs the `latest` tag. Typically, projects only use the `latest` tag for stable release versions, and use other tags for unstable versions such as prereleases.

The `next` tag is used by some projects to identify the upcoming version.

By default, other than `latest`, no tag has any special significance to npm itself.

# CAVEATS

This command used to be known as `npm tag`, which only created new tags, and so had a different syntax.

Tags must share a namespace with version numbers, because they are specified in the same slot: `npm install <pkg>@<version>` vs `npm install <pkg>@<tag>`.

Tags that can be interpreted as valid semver ranges will be rejected. For example, `v1.4` cannot be used as a tag, because it is interpreted by semver as `>=1.4.0 <1.5.0`. See <https://github.com/npm/npm/issues/6082>.

The simplest way to avoid semver problems with tags is to use tags that do not begin with a number or the letter `v`.

## See Also

- [npm-tag](#)
- [npm-publish](#)
- [npm-install](#)
- [npm-dedupe](#)
- [npm-registry](#)
- [npm-config](#)
- [npm-config](#)
- [npmrc](#)

---

Last modified January 21, 2016

Found a typo? Send a [pull request!](#)

## npm-docs

## Synopsis

```
npm docs [<pkgname> [<pkgname> ...]]
npm docs .
npm home [<pkgname> [<pkgname> ...]]
npm home .
```

## Description

This command tries to guess at the likely location of a package's documentation URL, and then tries to open it using the `--browser` config param. You can pass multiple package names at once. If no package name is provided, it will search for a `package.json` in the current folder and use the `name` property.

## Configuration

### browser

- Default: OS X: `"open"`, Windows: `"start"`, Others: `"xdg-open"`
- Type: String

The browser that is called by the `npm docs` command to open websites.

### registry

- Default: <https://registry.npmjs.org/>
- Type: url

The base URL of the npm package registry.

## See Also

- [npm-view](#)
- [npm-publish](#)



- [npm-registry](#)
- [npm-config](#)
- [npm-config](#)
- [npmrc](#)
- [package.json](#)

---

Last modified January 08, 2016      Found a typo? Send a [pull request!](#)

## npm-edit

# Synopsis

```
npm edit <pkg>[@<version>]
```

## Description

Opens the package folder in the default editor (or whatever you've configured as the npm **editor** config -- see [npm-config](#) .)

After it has been edited, the package is rebuilt so as to pick up any changes in compiled packages.

For instance, you can do **npm install connect** to install connect into your package, and then **npm edit connect** to make a few changes to your locally installed copy.

## Configuration

### editor

- Default: **EDITOR** environment variable if set, or **"vi"** on Posix, or **"notepad"** on Windows.
- Type: path

The command to run for **npm edit** or **npm config edit** .

## See Also

- [npm-folders](#)
- [npm-explore](#)
- [npm-install](#)
- [npm-config](#)
- [npm-config](#)
- [npmrc](#)

---

Last modified January 08, 2016      Found a typo? Send a [pull request!](#)

## npm-explore

# Synopsis

```
npm explore <pkg> [ -- <cmd>]
```

## Description

Spawn a subshell in the directory of the installed package specified.

If a command is specified, then it is run in the subshell, which then immediately terminates.

This is particularly handy in the case of git submodules in the `node_modules` folder:

```
npm explore some-dependency -- git pull origin master
```

Note that the package is *not* automatically rebuilt afterwards, so be sure to use `npm rebuild <pkg>` if you make any changes.

## Configuration

### shell

- Default: SHELL environment variable, or "bash" on Posix, or "cmd" on Windows
- Type: path

The shell to run for the `npm explore` command.

## See Also

- [npm-folders](#)
- [npm-edit](#)
- [npm-rebuild](#)
- [npm-build](#)
- [npm-install](#)

---

Last modified January 08, 2016

Found a typo? Send a [pull request!](#)

## npm-help

## Synopsis

```
npm help <term> [<terms...>]
```

## Description

If supplied a topic, then show the appropriate documentation page.

If the topic does not exist, or if multiple terms are provided, then run the `help-search` command to find a match. Note that, if `help-search` finds a single subject, then it will run `help` on that topic, so unique matches are equivalent to specifying a topic name.

## Configuration

### viewer

- Default: "man" on Posix, "browser" on Windows
- Type: path

The program to use to view help content.

Set to **"browser"** to view html help content in the default web browser.

## See Also

- [npm](#)
- README
- [npm-faq](#)
- [npm-folders](#)
- [npm-config](#)
- [npm-config](#)
- [npmrc](#)
- [package.json](#)
- [npm-help-search](#)
- [npm-index](#)

---

Last modified January 08, 2016

Found a typo? Send a [pull request!](#)

## npm-help-search

## Synopsis

```
npm help-search <text>
```

## Description

This command will search the npm markdown documentation files for the terms provided, and then list the results, sorted by relevance.

If only one result is found, then it will show that help topic.

If the argument to **npm help** is not a known help topic, then it will call **help-search** . It is rarely if ever necessary to call this command directly.

## Configuration

### long

- Type: Boolean
- Default false

If true, the "long" flag will cause help-search to output context around where the terms were found in the documentation.

If false, then help-search will just list out the help topics found.

## See Also

- [npm](#)
- [npm-faq](#)
- [npm-help](#)

## npm-init

# Synopsis

```
npm init [-f|--force|-y|--yes]
```

## Description

This will ask you a bunch of questions, and then write a package.json for you.

It attempts to make reasonable guesses about what you want things to be set to, and then writes a package.json file with the options you've selected.

If you already have a package.json file, it'll read that first, and default to the options in there.

It is strictly additive, so it does not delete options from your package.json without a really good reason to do so.

If you invoke it with `-f` , `--force` , `-y` , or `--yes` , it will use only defaults and not prompt you for any options.

## Configuration

### scope

- Default: none
- Type: String

The scope under which the new module should be created.

## See Also

- <https://github.com/isaacs/init-package-json>
- [package.json](#)
- [npm-version](#)
- [npm-scope](#)

## npm-install

# Synopsis

```
npm install (with no args, in package dir)
npm install [<@scope>/]<name>
npm install [<@scope>/]<name>@<tag>
npm install [<@scope>/]<name>@<version>
npm install [<@scope>/]<name>@<version range>
npm install <tarball file>
npm install <tarball url>
npm install <folder>
```

alias: `npm i`

common options: `[-S|--save|-D|--save-dev|-O|--save-optional] [-E|--save-exact] [--dry-run]`

# Description

This command installs a package, and any packages that it depends on. If the package has a shrinkwrap file, the installation of dependencies will be driven by that. See [npm-shrinkwrap](#).

A **package** is:

- a) a folder containing a program described by a [package.json](#) file
- b) a gzipped tarball containing (a)
- c) a url that resolves to (b)
- d) a `<name>@<version>` that is published on the registry (see [npm-registry](#)) with (c)
- e) a `<name>@<tag>` (see [npm-dist-tag](#)) that points to (d)
- f) a `<name>` that has a "latest" tag satisfying (e)
- g) a `<git remote url>` that resolves to (a)

Even if you never publish your package, you can still get a lot of benefits of using npm if you just want to write a node program (a), and perhaps if you also want to be able to easily install it elsewhere after packing it up into a tarball (b).

- **npm install** (in package directory, no arguments):

Install the dependencies in the local node\_modules folder.

In global mode (ie, with `-g` or `--global` appended to the command), it installs the current package context (ie, the current working directory) as a global package.

By default, **npm install** will install all modules listed as dependencies in [package.json](#).

With the `--production` flag (or when the `NODE_ENV` environment variable is set to `production`), npm will not install modules listed in `devDependencies`.

- **npm install <folder>** :

Install a package that is sitting in a folder on the filesystem.

- **npm install <tarball file>** :

Install a package that is sitting on the filesystem. Note: if you just want to link a dev directory into your npm root, you can do this more easily by using **npm link**.

Example:

```
npm install ./package.tgz
```

- **npm install <tarball url>** :

Fetch the tarball url, and then install it. In order to distinguish between this and other options, the argument must start with "http://" or "https://"

Example:

```
npm install https://github.com/indexzero/forever/tarball/v0.5.6
```

- **npm install [<@scope>]<name> [-S|--save|-D|--save-dev|-O|--save-optional]** :

Do a `<name>@<tag>` install, where `<tag>` is the "tag" config. (See [npm-config](#). The config's default value is `latest`.)

In most cases, this will install the latest version of the module published on npm.

Example:

```
npm install sax
```

**npm install** takes 3 exclusive, optional flags which save or update the package version in your main package.json:

- **-S, --save** : Package will appear in your **dependencies** .
- **-D, --save-dev** : Package will appear in your **devDependencies** .
- **-O, --save-optional** : Package will appear in your **optionalDependencies** .

When using any of the above options to save dependencies to your package.json, there is an additional, optional flag:

- **-E, --save-exact** : Saved dependencies will be configured with an exact version rather than using npm's default semver range operator.

Further, if you have an **npm-shrinkwrap.json** then it will be updated as well.

**<scope>** is optional. The package will be downloaded from the registry associated with the specified scope. If no registry is associated with the given scope the default registry is assumed. See [npm-scope](#) .

Note: if you do not include the @-symbol on your scope name, npm will interpret this as a GitHub repository instead, see below. Scopes names must also be followed by a slash.

Examples:

```
npm install sax --save
npm install githubname/reponame
npm install @myorg/privatepackage
npm install node-tap --save-dev
npm install dtrace-provider --save-optional
npm install readable-stream --save --save-exact
```

**\*\*Note\*\***: If there is a file or folder named `<name>` in the current working directory, then it will try to install that, and only try to fetch the package by name if it is not valid.

- **npm install [<@scope>/]<name>@<tag>** :

Install the version of the package that is referenced by the specified tag. If the tag does not exist in the registry data for that package, then this will fail.

Example:

```
npm install sax@latest
npm install @myorg/mypackage@latest
```

- **npm install [<@scope>/]<name>@<version>** :

Install the specified version of the package. This will fail if the version has not been published to the registry.

Example:

```
npm install sax@0.1.1
npm install @myorg/privatepackage@1.5.0
```

- **npm install [<@scope>/]<name>@<version range>** :

Install a version of the package matching the specified version range. This will follow the same rules for resolving dependencies described in [package.json](#) .

Note that most version ranges must be put in quotes so that your shell will treat it as a single argument.

Example:

```
npm install sax@">=0.1.0 <0.2.0"
npm install @myorg/privatepackage@">=0.1.0 <0.2.0"
```

- **npm install <git remote url> :**

Installs the package from the hosted git provider, cloning it with **git** . First it tries via the https (git with github) and if that fails, via ssh.

```
<protocol>://[<user>[:<password>]@]<hostname>[:<port>][:]/<path>[#<commit-ish>]
```

**<protocol>** is one of **git** , **git+ssh** , **git+http** , **git+https** , or **git+file** . If no **<commit-ish>** is specified, then **master** is used.

If the repository makes use of submodules, those submodules will be cloned as well.

The following git environment variables are recognized by npm and will be added to the environment when running git:

- **GIT\_ASKPASS**
- **GIT\_PROXY\_COMMAND**
- **GIT\_SSH**
- **GIT\_SSH\_COMMAND**
- **GIT\_SSL\_CAINFO**
- **GIT\_SSL\_NO\_VERIFY**

See the git man page for details.

Examples:

```
npm install git+ssh://git@github.com:npm/npm.git#v1.0.27
npm install git+https://isaacs@github.com:npm/npm.git
npm install git://github.com:npm/npm.git#v1.0.27
GIT_SSH_COMMAND='ssh -i ~/.ssh/custom_ident' npm install git+ssh://git@github.com:npm/npm.git
```

- **npm install <githubname>/<githubrepo>[#<commit-ish>] :**
- **npm install github:<githubname>/<githubrepo>[#<commit-ish>] :**

Install the package at **https://github.com/githubname/githubrepo** by attempting to clone it using **git** .

If you don't specify a *commit-ish* then **master** will be used.

Examples:

```
npm install mygithubuser/myproject
npm install github:mygithubuser/myproject
```

- **npm install gist:[<githubname>]/<gistID>[#<commit-ish>] :**

Install the package at **https://gist.github.com/gistID** by attempting to clone it using **git** . The GitHub username associated with the gist is optional and will not be saved in **package.json** if **-S** or **--save** is used.

If you don't specify a *commit-ish* then **master** will be used.

Example:

```
npm install gist:101a11beef
```

- **npm install bitbucket:<bitbucketname>/<bitbucketrepo>[#<commit-ish>] :**

Install the package at **https://bitbucket.org/bitbucketname/bitbucketrepo** by attempting to clone it using **git** .

If you don't specify a *commit-ish* then **master** will be used.

Example:

```
npm install bitbucket:mybitbucketuser/myproject
```

- `npm install gitlab:<gitlabname>/<gitlabrepo>[#<commit-ish>]` :

Install the package at `https://gitlab.com/gitlabname/gitlabrepo` by attempting to clone it using `git` .

If you don't specify a *commit-ish* then `master` will be used.

Example:

```
npm install gitlab:mygitlabuser/myproject
```

You may combine multiple arguments, and even multiple types of arguments. For example:

```
npm install sax@">=0.1.0 <0.2.0" bench supervisor
```

The `--tag` argument will apply to all of the specified install targets. If a tag with the given name exists, the tagged version is preferred over newer versions.

The `--dry-run` argument will report in the usual way what the install would have done without actually installing anything.

The `-f` or `--force` argument will force npm to fetch remote resources even if a local copy exists on disk.

```
npm install sax --force
```

The `-g` or `--global` argument will cause npm to install the package globally rather than locally. See [npm-folders](#) .

The `--global-style` argument will cause npm to install the package into your local `node_modules` folder with the same layout it uses with the global `node_modules` folder. Only your direct dependencies will show in `node_modules` and everything they depend on will be flattened in their `node_modules` folders. This obviously will eliminate some deduping.

The `--legacy-bundling` argument will cause npm to install the package such that versions of npm prior to 1.4, such as the one included with node 0.8, can install the package. This eliminates all automatic deduping.

The `--link` argument will cause npm to link global installs into the local space in some cases.

The `--no-bin-links` argument will prevent npm from creating symlinks for any binaries the package might contain.

The `--no-optional` argument will prevent optional dependencies from being installed.

The `--no-shrinkwrap` argument, which will ignore an available shrinkwrap file and use the package.json instead.

The `--nodedir=/path/to/node/source` argument will allow npm to find the node source code so that npm can compile native modules.

The `--only={prod[uction]|dev[elopment]}` argument will cause either only `devDependencies` or only non- `devDependencies` to be installed regardless of the `NODE_ENV` .

See [npm-config](#) . Many of the configuration params have some effect on installation, since that's most of what npm does.

## ALGORITHM

To install a package, npm uses the following algorithm:

```
load the existing node_modules tree from disk
clone the tree
fetch the package.json and assorted metadata and add it to the clone
walk the clone and add any missing dependencies
  dependencies will be added as close to the top as is possible
  without breaking any other modules
compare the original tree with the cloned tree and make a list of
actions to take to convert one to the other
execute all of the actions, deepest first
  kinds of actions are install, update, remove and move
```

For this `package{dep}` structure: `A{B,C}`, `B{C}`, `C{D}` ,this algorithm produces:



```

A
+-- B
+-- C
+-- D

```

That is, the dependency from B to C is satisfied by the fact that A already caused C to be installed at a higher level. D is still installed at the top level because nothing conflicts with it.

For `A{B,C}`, `B{C,D@1}`, `C{D@2}`, this algorithm produces:

```

A
+-- B
+-- C
   |-- D@2
   +-- D@1

```

Because B's D@1 will be installed in the top level, C now has to install D@2 privately for itself.

See [npm-folders](#) for a more detailed description of the specific folder structures that npm creates.

### Limitations of npm's Install Algorithm

There are some very rare and pathological edge-cases where a cycle can cause npm to try to install a never-ending tree of packages. Here is the simplest case:

```
A -> B -> A' -> B' -> A -> B -> A' -> B' -> A -> ...
```

where `A` is some version of a package, and `A'` is a different version of the same package. Because `B` depends on a different version of `A` than the one that is already in the tree, it must install a separate copy. The same is true of `A'`, which must install `B'`. Because `B'` depends on the original version of `A`, which has been overridden, the cycle falls into infinite regress.

To avoid this situation, npm flat-out refuses to install any `name@version` that is already present anywhere in the tree of package folder ancestors. A more correct, but more complex, solution would be to symlink the existing version into the new location. If this ever affects a real use-case, it will be investigated.

## See Also

- [npm-folders](#)
- [npm-update](#)
- [npm-link](#)
- [npm-rebuild](#)
- [npm-scripts](#)
- [npm-build](#)
- [npm-config](#)
- [npm-config](#)
- [npmrc](#)
- [npm-registry](#)
- [npm-tag](#)
- [npm-uninstall](#)
- [npm-shrinkwrap](#)
- [package.json](#)

---

Last modified January 29, 2016

Found a typo? Send a [pull request!](#)

## npm install-test -- Install package(s) and run tests

### Synopsis

```

npm install-test (with no args, in package dir)
npm install-test [<@scope>/]<name>
npm install-test [<@scope>/]<name>@<tag>
npm install-test [<@scope>/]<name>@<version>
npm install-test [<@scope>/]<name>@<version range>
npm install-test <tarball file>
npm install-test <tarball url>
npm install-test <folder>

alias: npm it
common options: [--save|--save-dev|--save-optional] [--save-exact] [--dry-run]

```

## Description

This command runs an `npm install` followed immediately by an `npm test`. It takes exactly the same arguments as `npm install`.

## See Also

- [npm-install](#)
- [npm-test](#)

---

Last modified January 08, 2016    Found a typo? Send a [pull request!](#)

## npm-link

## Synopsis

```

npm link (in package dir)
npm link [<@scope>/]<pkg>[@<version>]

alias: npm ln

```

## Description

Package linking is a two-step process.

First, `npm link` in a package folder will create a globally-installed symbolic link from `prefix/package-name` to the current folder (see [npm-config](#) for the value of `prefix`).

Next, in some other location, `npm link package-name` will create a symlink from the local `node_modules` folder to the global symlink.

Note that `package-name` is taken from `package.json`, not from directory name.

The package name can be optionally prefixed with a scope. See [npm-scope](#). The scope must be preceded by an @-symbol and followed by a slash.

When creating tarballs for `npm publish`, the linked packages are "snapshotted" to their current state by resolving the symbolic links.

This is handy for installing your own stuff, so that you can work on it and test it iteratively without having to continually rebuild.

For example:

```

cd ~/projects/node-redis      # go into the package directory
npm link                     # creates global link
cd ~/projects/node-bloggy    # go into some other package directory.
npm link redis                # link-install the package

```

Now, any changes to `~/projects/node-redis` will be reflected in `~/projects/node-bloggy/node_modules/node-redis/`. Note that the link should be to the package name, not the directory name for that package.

You may also shortcut the two steps in one. For example, to do the above use-case in a shorter way:

```
cd ~/projects/node-bloggy # go into the dir of your main project
npm link ../node-redis    # link the dir of your dependency
```

The second line is the equivalent of doing:

```
(cd ../node-redis; npm link)
npm link node-redis
```

That is, it first creates a global link, and then links the global installation target into your project's `node_modules` folder.

If your linked package is scoped (see [npm-scope](#)) your link command must include that scope, e.g.

```
npm link @myorg/privatepackage
```

## See Also

- [npm-developers](#)
- [npm-faq](#)
- [package.json](#)
- [npm-install](#)
- [npm-folders](#)
- [npm-config](#)
- [npm-config](#)
- [npmrc](#)

---

Last modified January 08, 2016

Found a typo? Send a [pull request!](#)

## npm-logout

## Synopsis

```
npm logout [--registry=<url>] [--scope=<@scope>]
```

## Description

When logged into a registry that supports token-based authentication, tell the server to end this token's session. This will invalidate the token everywhere you're using it, not just for the current environment.

When logged into a legacy registry that uses username and password authentication, this will clear the credentials in your user configuration. In this case, it will *only* affect the current environment.

If `--scope` is provided, this will find the credentials for the registry connected to that scope, if set.

## Configuration

### registry

Default: <https://registry.npmjs.org/>

The base URL of the npm package registry. If `scope` is also specified, it takes precedence.

### scope

Default: none

If specified, the user and login credentials given will be associated with the specified scope. See [npm-scope](#). You can use both at the same time, e.g.

```
npm adduser --registry=http://myregistry.example.com --scope=@myco
```

This will set a registry for the given scope and login or create a user for that registry at the same time.

## See Also

- [npm-adduser](#)
- [npm-registry](#)
- [npm-config](#)
- [npm-config](#)
- [npmrc](#)
- [npm-whoami](#)

---

Last modified January 08, 2016

Found a typo? Send a [pull request!](#)

## npm-ls

## Synopsis

```
npm ls [[<@scope>/]<pkg> ...]  
aliases: list, la, ll
```

## Description

This command will print to stdout all the versions of packages that are installed, as well as their dependencies, in a tree-structure.

Positional arguments are **name@version-range** identifiers, which will limit the results to only the paths to the packages named. Note that nested packages will *also* show the paths to the specified packages. For example, running **npm ls promzard** in npm's source tree will show:

```
npm@VERSION@ /path/to/npm  
├── init-package-json@0.0.4  
└── promzard@0.1.5
```

It will print out extraneous, missing, and invalid packages.

If a project specifies git urls for dependencies these are shown in parentheses after the name@version to make it easier for users to recognize potential forks of a project.

The tree shown is the logical dependency tree, based on package dependencies, not the physical layout of your node\_modules folder.

When run as **ll** or **la**, it shows extended information by default.

## Configuration

json

- Default: false
- Type: Boolean

Show information in JSON format.

### long

- Default: false
- Type: Boolean

Show extended information.

### parseable

- Default: false
- Type: Boolean

Show parseable output instead of tree view.

### global

- Default: false
- Type: Boolean

List packages in the global install prefix instead of in the current project.

### depth

- Type: Int

Max display depth of the dependency tree.

### prod / production

- Type: Boolean
- Default: false

Display only the dependency tree for packages in **dependencies** .

### dev

- Type: Boolean
- Default: false

Display only the dependency tree for packages in **devDependencies** .

### only

- Type: String

When "dev" or "development", is an alias to **dev** .

When "prod" or "production", is an alias to **production** .`

## See Also

- [npm-config](#)
- [npm-config](#)
- [npmrc](#)
- [npm-folders](#)
- [npm-install](#)
- [npm-link](#)
- [npm-prune](#)
- [npm-outdated](#)
- [npm-update](#)

# npm

## Synopsis

```
npm <command> [args]
```

## VERSION

@VERSION@

## Description

npm is the package manager for the Node JavaScript platform. It puts modules in place so that node can find them, and manages dependency conflicts intelligently.

It is extremely configurable to support a wide variety of use cases. Most commonly, it is used to publish, discover, install, and develop node programs.

Run `npm help` to get a list of available commands.

## INTRODUCTION

You probably got npm because you want to install stuff.

Use `npm install blerg` to install the latest version of "blerg". Check out [npm-install](#) for more info. It can do a lot of stuff.

Use the `npm search` command to show everything that's available. Use `npm ls` to show everything you've installed.

## DEPENDENCIES

If a package references to another package with a git URL, npm depends on a preinstalled git.

If one of the packages npm tries to install is a native node module and requires compiling of C++ Code, npm will use [node-gyp](#) for that task. For a Unix system, [node-gyp](#) needs Python, make and a buildchain like GCC. On Windows, Python and Microsoft Visual Studio C++ is needed. Python 3 is not supported by [node-gyp](#). For more information visit [the node-gyp repository](#) and the [node-gyp Wiki](#).

## DIRECTORIES

See [npm-folders](#) to learn about where npm puts stuff.

In particular, npm has two modes of operation:

- global mode:  
npm installs packages into the install prefix at `prefix/lib/node_modules` and bins are installed in `prefix/bin`.
- local mode:  
npm installs packages into the current project directory, which defaults to the current working directory. Packages are installed to `./node_modules`, and bins are installed to `./node_modules/.bin`.

Local mode is the default. Use `-g` or `--global` on any command to operate in global mode instead.

## DEVELOPER USAGE

If you're using npm to develop and publish your code, check out the following help topics:

- `json`: Make a `package.json` file. See [package.json](#) .
- `link`: For linking your current working code into Node's path, so that you don't have to reinstall every time you make a change. Use `npm link` to do this.
- `install`: It's a good idea to install things if you don't need the symbolic link. Especially, installing other peoples code from the registry is done via `npm install`
- `adduser`: Create an account or log in. Credentials are stored in the user config file.
- `publish`: Use the `npm publish` command to upload your code to the registry.

## Configuration

npm is extremely configurable. It reads its configuration options from 5 places.

- **Command line switches:**  
Set a config with `--key val` . All keys take a value, even if they are booleans (the config parser doesn't know what the options are at the time of parsing.) If no value is provided, then the option is set to boolean `true` .
- **Environment Variables:**  
Set any config by prefixing the name in an environment variable with `npm_config_` . For example, `export npm_config_key=val` .
- **User Configs:**  
The file at `$HOME/.npmrc` is an ini-formatted list of configs. If present, it is parsed. If the `userconfig` option is set in the cli or env, then that will be used instead.
- **Global Configs:**  
The file found at `./etc/npmrc` (from the node executable, by default this resolves to `/usr/local/etc/npmrc`) will be parsed if it is found. If the `globalconfig` option is set in the cli, env, or user config, then that file is parsed instead.
- **Defaults:**  
npm's default configuration options are defined in `lib/utils/config-defs.js`. These must not be changed.

See [npm-config](#) for much much more information.

## CONTRIBUTIONS

Patches welcome!

- `code`: Read through [npm-coding-style](#) if you plan to submit code. You don't have to agree with it, but you do have to follow it.
- `docs`: If you find an error in the documentation, edit the appropriate markdown file in the "doc" folder. (Don't worry about generating the man page.)

Contributors are listed in npm's `package.json` file. You can view them easily by doing `npm view npm contributors` .

If you would like to contribute, but don't know what to work on, check the issues list or ask on the mailing list.

- <https://github.com/npm/npm/issues>
- [npm-@googlegroups.com](mailto:npm-@googlegroups.com)

## BUGS

When you find issues, please report them:

- web: <https://github.com/npm/npm/issues>
- email: [npm-@googlegroups.com](mailto:npm-@googlegroups.com)

Be sure to include *all* of the output from the npm command that didn't work as expected. The `npm-debug.log` file is also helpful to provide.

You can also look for isaacs in `#node.js` on `irc://irc.freenode.net`. He will no doubt tell you to put the output in a gist or email.

## AUTHOR

[Isaac Z. Schlueter](#) :: [isaacs](#) :: [@izs](#) :: [i@izs.me](#)

## See Also

- [npm-help](#)
- [npm-faq](#)
- [README](#)
- [package.json](#)
- [npm-install](#)
- [npm-config](#)
- [npm-config](#)
- [npmrc](#)
- [npm-index](#)

Last modified January 08, 2016

Found a typo? Send a [pull request!](#)

## npm-outdated

# Synopsis

```
npm outdated [[<@scope>/]<pkg> ...]
```

## Description

This command will check the registry to see if any (or, specific) installed packages are currently outdated.

In the output:

- **wanted** is the maximum version of the package that satisfies the semver range specified in **package.json** . If there's no available semver range (i.e. you're running **npm outdated --global** , or the package isn't included in **package.json** ), then **wanted** shows the currently-installed version.
- **latest** is the version of the package tagged as latest in the registry. Running **npm publish** with no special configuration will publish the package with a dist-tag of **latest** . This may or may not be the maximum version of the package, or the most-recently published version of the package, depending on how the package's developer manages the latest [dist-tag](#).
- **location** is where in the dependency tree the package is located. Note that **npm outdated** defaults to a depth of 0, so unless you override that, you'll always be seeing only top-level dependencies that are outdated.
- **package type** (when using **--long / -l** ) tells you whether this package is a **dependency** or a **devDependency** . Packages not included in **package.json** are always marked **dependencies** .

### An example

```
$ npm outdated
Package      Current   Wanted   Latest   Location
glob         5.0.15   5.0.15   6.0.1    test-outdated-output
nothingness  0.0.3    git      git      test-outdated-output
npm          3.5.1    3.5.2    3.5.1    test-outdated-output
local-dev    0.0.3    linked   linked   test-outdated-output
once         1.3.2    1.3.3    1.3.3    test-outdated-output
```

With these **dependencies** :

```
{
  "glob": "^5.0.15",
  "nothingness": "github:othiym23/nothingness#master",
  "npm": "^3.5.1",
  "once": "^1.3.1"
}
```

A few things to note:

- **glob** requires **^5** , which prevents npm from installing **glob@6** , which is outside the semver range.
- Git dependencies will always be reinstalled, because of how they're specified. The installed committish might satisfy the dependency specifier (if it's something immutable, like a commit SHA), or it might not, so **npm outdated** and **npm update** have to fetch Git repos to check. This is why currently doing a reinstall of a Git dependency always forces a new clone and install.
- **npm@3.5.2** is marked as "wanted", but "latest" is **npm@3.5.1** because npm uses dist-tags to manage its **latest** and **next** release channels. **npm update** will install the *newest* version, but **npm install npm** (with no semver range) will install whatever's tagged as **latest** .



- `once` is just plain out of date. Reinstalling `node_modules` from scratch or running `npm update` will bring it up to spec.

## Configuration

### `json`

- Default: false
- Type: Boolean

Show information in JSON format.

### `long`

- Default: false
- Type: Boolean

Show extended information.

### `parseable`

- Default: false
- Type: Boolean

Show parseable output instead of tree view.

### `global`

- Default: false
- Type: Boolean

Check packages in the global install prefix instead of in the current project.

### `depth`

- Default: 0
- Type: Int

Max depth for checking dependency tree.

## See Also

- [npm-update](#)
- [npm-dist-tag](#)
- [npm-registry](#)
- [npm-folders](#)

---

Last modified January 21, 2016

Found a typo? Send a [pull request!](#)

## npm-owner

## Synopsis

```
npm owner add <user> [<@scope>/]<pkg>
npm owner rm <user> [<@scope>/]<pkg>
npm owner ls [<@scope>/]<pkg>
```

# Description

Manage ownership of published packages.

- `ls`: List all the users who have access to modify a package and push new versions. Handy when you need to know who to bug for help.
- `add`: Add a new user as a maintainer of a package. This user is enabled to modify metadata, publish new versions, and add other owners.
- `rm`: Remove a user from the package owner list. This immediately revokes their privileges.

Note that there is only one level of access. Either you can modify a package, or you can't. Future versions may contain more fine-grained access levels, but that is not implemented at this time.

## See Also

- [npm-publish](#)
- [npm-registry](#)
- [npm-adduser](#)
- [npm-disputes](#)

---

Last modified January 08, 2016

Found a typo? Send a [pull request!](#)

## npm-pack

## Synopsis

```
npm pack [[<@scope>/]<pkg>...]
```

## Description

For anything that's installable (that is, a package folder, tarball, tarball url, name@tag, name@version, name, or scoped name), this command will fetch it to the cache, and then copy the tarball to the current working directory as `<name>-<version>.tgz`, and then write the filenames out to stdout.

If the same package is specified multiple times, then the file will be overwritten the second time.

If no arguments are supplied, then npm packs the current package folder.

## See Also

- [npm-cache](#)
- [npm-publish](#)
- [npm-config](#)
- [npm-config](#)
- [npmrc](#)

---

Last modified January 08, 2016

Found a typo? Send a [pull request!](#)

## npm-ping

# Synopsis

```
npm ping [--registry <registry>]
```

## Description

Ping the configured or given npm registry and verify authentication.

## See Also

- [npm-config](#)
- [npm-config](#)
- [npmrc](#)

---

Last modified January 08, 2016

Found a typo? Send a [pull request!](#)

## npm-prefix

# Synopsis

```
npm prefix [-g]
```

## Description

Print the local prefix to standard out. This is the closest parent directory to contain a package.json file unless `-g` is also specified.

If `-g` is specified, this will be the value of the global prefix. See [npm-config](#) for more detail.

## See Also

- [npm-root](#)
- [npm-bin](#)
- [npm-folders](#)
- [npm-config](#)
- [npm-config](#)
- [npmrc](#)

---

Last modified January 08, 2016

Found a typo? Send a [pull request!](#)

## npm-prune

# Synopsis

```
npm prune [[<@scope>/]<pkg>...] [--production]
```

## Description

This command removes "extraneous" packages. If a package name is provided, then only packages matching one of the supplied names are removed.

Extraneous packages are packages that are not listed on the parent package's dependencies list.

If the `--production` flag is specified or the `NODE_ENV` environment variable is set to `production`, this command will remove the packages specified in your `devDependencies`. Setting `--production=false` will negate `NODE_ENV` being set to `production`.

## See Also

- [npm-uninstall](#)
- [npm-folders](#)
- [npm-ls](#)

---

Last modified January 08, 2016      Found a typo? Send a [pull request!](#)

## npm-publish

## Synopsis

```
npm publish [<tarball>|<folder>] [--tag <tag>] [--access <public|restricted>]
```

Publishes '.' if no argument supplied  
Sets tag 'latest' if no --tag specified

## Description

Publishes a package to the registry so that it can be installed by name. All files in the package directory are included if no local `.gitignore` or `.npmignore` file exists. If both files exist and a file is ignored by `.gitignore` but not by `.npmignore` then it will be included. See [npm-developers](#) for full details on what's included in the published package, as well as details on how the package is built.

By default npm will publish to the public registry. This can be overridden by specifying a different default registry or using a [npm-scope](#) in the name (see [package.json](#)).

- `<folder>` : A folder containing a package.json file
- `<tarball>` : A url or file path to a gzipped tar archive containing a single folder with a package.json file inside.
- `[--tag <tag>]` Registers the published package with the given tag, such that `npm install <name>@<tag>` will install this version. By default, `npm publish` updates and `npm install` installs the `latest` tag. See [npm-dist-tag](#) for details about tags.
- `[--access <public|restricted>]` Tells the registry whether this package should be published as public or restricted. Only applies to scoped packages, which default to `restricted`. If you don't have a paid account, you must publish with `--access public` to publish scoped packages.

Fails if the package name and version combination already exists in the specified registry.

Once a package is published with a given name and version, that specific name and version combination can never be used again, even if it is removed with [npm-unpublish](#).

## See Also

- [npm-registry](#)
- [npm-scope](#)
- [npm-adduser](#)
- [npm-owner](#)
- [npm-deprecate](#)
- [npm-tag](#)

---

Last modified January 21, 2016

Found a typo? Send a [pull request!](#)

## npm-rebuild

## Synopsis

```
npm rebuild [ [<@scope>/<name>]... ]  
alias: npm rb
```

## Description

This command runs the **npm build** command on the matched folders. This is useful when you install a new version of node, and must recompile all your C++ addons with the new binary.

## See Also

- [npm-build](#)
- [npm-install](#)

---

Last modified January 08, 2016

Found a typo? Send a [pull request!](#)

## npm-repo

## Synopsis

```
npm repo [<pkg>]
```

## Description

This command tries to guess at the likely location of a package's repository URL, and then tries to open it using the **--browser** config param. If no package name is provided, it will search for a **package.json** in the current folder and use the **name** property.

## Configuration

## browser

- Default: OS X: `"open"` , Windows: `"start"` , Others: `"xdg-open"`
- Type: String

The browser that is called by the `npm repo` command to open websites.

## See Also

- [npm-docs](#)
- [npm-config](#)

---

Last modified January 08, 2016

Found a typo? Send a [pull request!](#)

## npm-restart

## Synopsis

```
npm restart [-- <args>]
```

## Description

This restarts a package.

This runs a package's "stop", "restart", and "start" scripts, and associated pre- and post- scripts, in the order given below:

1. prerestart
2. prestop
3. stop
4. poststop
5. restart
6. prestart
7. start
8. poststart
9. postrestart

## NOTE

Note that the "restart" script is run **in addition to** the "stop" and "start" scripts, not instead of them.

This is the behavior as of `npm` major version 2. A change in this behavior will be accompanied by an increase in major version number

## See Also

- [npm-run-script](#)
- [npm-scripts](#)
- [npm-test](#)
- [npm-start](#)
- [npm-stop](#)
- [npm-restart](#)

---

Last modified January 08, 2016Found a typo? Send a [pull request!](#)

## npm-root

# Synopsis

```
npm root [-g]
```

## Description

Print the effective `node_modules` folder to standard out.

## See Also

- [npm-prefix](#)
- [npm-bin](#)
- [npm-folders](#)
- [npm-config](#)
- [npm-config](#)
- [npmrc](#)

---

Last modified January 08, 2016Found a typo? Send a [pull request!](#)

## npm-run-script

# Synopsis

```
npm run-script <command> [-- <args>...]
```

```
alias: npm run
```

## Description

This runs an arbitrary command from a package's `"scripts"` object. If no `"command"` is provided, it will list the available scripts. `run[-script]` is used by the test, start, restart, and stop commands, but can be called directly, as well. When the scripts in the package are printed out, they're separated into lifecycle (test, start, restart) and directly-run scripts.

As of [npm@2.0.0](#), you can use custom arguments when executing scripts. The special option `--` is used by [getopt](#) to delimit the end of the options. npm will pass all the arguments after the `--` directly to your script:

```
npm run test -- --grep="pattern"
```

The arguments will only be passed to the script specified after `npm run` and not to any pre or post script.

The `env` script is a special built-in command that can be used to list environment variables that will be available to the script at runtime. If an "env" command is defined in your package it will take precedence over the built-in.

In addition to the shell's pre-existing `PATH`, `npm run` adds `node_modules/.bin` to the `PATH` provided to scripts. Any binaries provided by locally-installed dependencies can be used without the `node_modules/.bin` prefix. For example, if there is a `devDependency` on `tap` in your package, you

should write:

```
"scripts": {"test": "tap test/*.js"}
```

instead of `"scripts": {"test": "node_modules/.bin/tap test/*.js"}` to run your tests.

If you try to run a script without having a `node_modules` directory and it fails, you will be given a warning to run `npm install`, just in case you've forgotten.

## See Also

- [npm-scripts](#)
- [npm-test](#)
- [npm-start](#)
- [npm-restart](#)
- [npm-stop](#)

---

Last modified January 21, 2016

Found a typo? Send a [pull request!](#)

## npm-search

## Synopsis

```
npm search [-l|--long] [search terms ...]
```

```
aliases: s, se
```

## Description

Search the registry for packages matching the search terms.

If a term starts with `/`, then it's interpreted as a regular expression. A trailing `/` will be ignored in this case. (Note that many regular expression characters must be escaped or quoted in most shells.)

## Configuration

### long

- Default: false
- Type: Boolean

Display full package descriptions and other long text across multiple lines. When disabled (default) search results are truncated to fit neatly on a single line. Modules with extremely long names will fall on multiple lines.

## See Also

- [npm-registry](#)
- [npm-config](#)
- [npm-config](#)
- [npmrc](#)
- [npm-view](#)



# npm-shrinkwrap

## Synopsis

```
npm shrinkwrap
```

## Description

This command locks down the versions of a package's dependencies so that you can control exactly which versions of each dependency will be used when your package is installed. The `package.json` file is still required if you want to use `npm install`.

By default, `npm install` recursively installs the target's dependencies (as specified in `package.json`), choosing the latest available version that satisfies the dependency's semver pattern. In some situations, particularly when shipping software where each change is tightly managed, it's desirable to fully specify each version of each dependency recursively so that subsequent builds and deploys do not inadvertently pick up newer versions of a dependency that satisfy the semver pattern. Specifying specific semver patterns in each dependency's `package.json` would facilitate this, but that's not always possible or desirable, as when another author owns the npm package. It's also possible to check dependencies directly into source control, but that may be undesirable for other reasons.

As an example, consider package A:

```
{
  "name": "A",
  "version": "0.1.0",
  "dependencies": {
    "B": "<0.1.0"
  }
}
```

package B:

```
{
  "name": "B",
  "version": "0.0.1",
  "dependencies": {
    "C": "<0.1.0"
  }
}
```

and package C:

```
{
  "name": "C",
  "version": "0.0.1"
}
```

If these are the only versions of A, B, and C available in the registry, then a normal `npm install A` will install:

```
A@0.1.0
├-- B@0.0.1
│  └-- C@0.0.1
```

However, if B@0.0.2 is published, then a fresh `npm install A` will install:

```
A@0.1.0
├-- B@0.0.2
│  └-- C@0.0.1
```

assuming the new version did not modify B's dependencies. Of course, the new version of B could include a new version of C and any number of new dependencies. If such changes are undesirable, the author of A could specify a dependency on B@0.0.1. However, if A's author and B's author are not the same person, there's no way for A's author to say that he or she does not want to pull in newly published versions of C when B hasn't changed at all.

In this case, A's author can run

```
npm shrinkwrap
```

This generates **npm-shrinkwrap.json**, which will look something like this:

```
{
  "name": "A",
  "version": "1.1.0",
  "dependencies": {
    "B": {
      "version": "1.0.1",
      "from": "B@^1.0.0",
      "resolved": "https://registry.npmjs.org/B/-/B-1.0.1.tgz",
      "dependencies": {
        "C": {
          "version": "1.0.1",
          "from": "org/C#v1.0.1",
          "resolved": "git://github.com/org/C.git#5c380ae319fc4efe9e7f2d9c78b0faa588fd99b4"
        }
      }
    }
  }
}
```

The shrinkwrap command has locked down the dependencies based on what's currently installed in **node\_modules**. The installation behavior is changed to:

1. The module tree described by the shrinkwrap is reproduced. This means reproducing the structure described in the file, using the specific files referenced in "resolved" if available, falling back to normal package resolution using "version" if one isn't.
2. The tree is walked and any missing dependencies are installed in the usual fashion.

## Using shrinkwrapped packages

Using a shrinkwrapped package is no different than using any other package: you can **npm install** it by hand, or add a dependency to your **package.json** file and **npm install** it.

## Building shrinkwrapped packages

To shrinkwrap an existing package:

1. Run **npm install** in the package root to install the current versions of all dependencies.
2. Validate that the package works as expected with these versions.
3. Run **npm shrinkwrap**, add **npm-shrinkwrap.json** to git, and publish your package.

To add or update a dependency in a shrinkwrapped package:

1. Run **npm install** in the package root to install the current versions of all dependencies.
2. Add or update dependencies. **npm install --save** each new or updated package individually to update the **package.json** and the shrinkwrap. Note that they must be explicitly named in order to be installed: running **npm install** with no arguments will merely reproduce the existing shrinkwrap.
3. Validate that the package works as expected with the new dependencies.
4. Commit the new **npm-shrinkwrap.json**, and publish your package.

You can use [npm-outdated](#) to view dependencies with newer versions available.

## Other Notes

A shrinkwrap file must be consistent with the package's **package.json** file. **npm shrinkwrap** will fail if required dependencies are not already installed, since that would result in a shrinkwrap that wouldn't actually work. Similarly, the command will fail if there are extraneous packages (not referenced by **package.json**), since that would indicate that **package.json** is not correct.

Since **npm shrinkwrap** is intended to lock down your dependencies for production use, **devDependencies** will not be included unless you explicitly set the **--dev** flag when you run **npm shrinkwrap**. If installed **devDependencies** are excluded, then npm will print a warning. If you want them to be installed with your module by default, please consider adding them to **dependencies** instead.

If shrinkwrapped package A depends on shrinkwrapped package B, B's shrinkwrap will not be used as part of the installation of A. However, because A's shrinkwrap is constructed from a valid installation of B and recursively specifies all dependencies, the contents of B's shrinkwrap will implicitly be included in A's shrinkwrap.

## Caveats

If you wish to lock down the specific bytes included in a package, for example to have 100% confidence in being able to reproduce a deployment or build, then you ought to check your dependencies into source control, or pursue some other mechanism that can verify contents rather than versions.

## See Also

- [npm-install](#)
- [package.json](#)
- [npm-ls](#)

---

Last modified January 08, 2016      Found a typo? Send a [pull request!](#)

## npm-star

## Synopsis

```
npm star [<pkg>...]
npm unstar [<pkg>...]
```

## Description

"Starring" a package means that you have some interest in it. It's a vaguely positive way to show that you care.

"Unstarring" is the same thing, but in reverse.

It's a boolean thing. Starring repeatedly has no additional effect.

## See Also

- [npm-view](#)
- [npm-whoami](#)
- [npm-adduser](#)

---

Last modified January 08, 2016      Found a typo? Send a [pull request!](#)

## npm-stars

## Synopsis

```
npm stars [<user>]
```

## Description

If you have starred a lot of neat things and want to find them again quickly this command lets you do just that.

You may also want to see your friend's favorite packages, in this case you will most certainly enjoy this command.

## See Also

- [npm-star](#)
- [npm-view](#)
- [npm-whoami](#)
- [npm-adduser](#)

---

Last modified January 08, 2016

Found a typo? Send a [pull request!](#)

## npm-start

## Synopsis

```
npm start [-- <args>]
```

## Description

This runs an arbitrary command specified in the package's **"start"** property of its **"scripts"** object. If no **"start"** property is specified on the **"scripts"** object, it will run **node server.js**.

As of [npm@2.0.0](#), you can use custom arguments when executing scripts. Refer to [npm-run-script](#) for more details.

## See Also

- [npm-run-script](#)
- [npm-scripts](#)
- [npm-test](#)
- [npm-restart](#)
- [npm-stop](#)

---

Last modified January 08, 2016

Found a typo? Send a [pull request!](#)

## npm-stop

## Synopsis

```
npm stop [-- <args>]
```

## Description

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

## See Also

- [npm-run-script](#)
- [npm-scripts](#)
- [npm-test](#)
- [npm-start](#)
- [npm-restart](#)

---

Last modified January 08, 2016

Found a typo? Send a [pull request!](#)

## npm-tag

# Synopsis

```
[DEPRECATED] npm tag <name>@<version> [<tag>]  
See `dist-tag`
```

## Description

THIS COMMAND IS DEPRECATED. See [npm-dist-tag](#) for details.

Tags the specified version of the package with the specified tag, or the `--tag` config if not specified.

A tag can be used when installing packages as a reference to a version instead of using a specific version number:

```
npm install <name>@<tag>
```

When installing dependencies, a preferred tagged version may be specified:

```
npm install --tag <tag>
```

This also applies to `npm dedupe` .

Publishing a package always sets the "latest" tag to the published version.

## PURPOSE

Tags can be used to provide an alias instead of version numbers. For example, `npm` currently uses the tag "next" to identify the upcoming version, and the tag "latest" to identify the current version.

A project might choose to have multiple streams of development, e.g., "stable", "canary".

## CAVEATS

Tags must share a namespace with version numbers, because they are specified in the same slot: `npm install <pkg>@<version>` vs `npm install <pkg>@<tag>` .

Tags that can be interpreted as valid semver ranges will be rejected. For example, `v1.4` cannot be used as a tag, because it is interpreted by semver as `>=1.4.0 <1.5.0` . See <https://github.com/npm/npm/issues/6082>.

The simplest way to avoid semver problems with tags is to use tags that do not begin with a number or the letter `v` .

## See Also

- [npm-publish](#)
- [npm-install](#)
- [npm-dedupe](#)
- [npm-registry](#)
- [npm-config](#)
- [npm-config](#)
- [npm-tag](#)
- [npmrc](#)

---

Last modified January 08, 2016      Found a typo? Send a [pull request!](#)

## npm-team

# Synopsis

```
npm team create <scope:team>
npm team destroy <scope:team>

npm team add <scope:team> <user>
npm team rm <scope:team> <user>

npm team ls <scope>|<scope:team>

npm team edit <scope:team>
```

# Description

Used to manage teams in organizations, and change team memberships. Does not handle permissions for packages.

Teams must always be fully qualified with the organization/scope they belong to when operating on them, separated by a colon ( `:` ). That is, if you have a **developers** team on a **foo** organization, you must always refer to that team as **foo:developers** in these commands.

- `create / destroy`: Create a new team, or destroy an existing one.
- `add / rm`: Add a user to an existing team, or remove a user from a team they belong to.
- `ls`: If performed on an organization name, will return a list of existing teams under that organization. If performed on a team, it will instead return a list of all users belonging to that particular team.

# DETAILS

**npm team** always operates directly on the current registry, configurable from the command line using `--registry=<registry url>` .

In order to create teams and manage team membership, you must be a *team admin* under the given organization. Listing teams and team memberships may be done by any member of the organizations.

Organization creation and management of team admins and *organization* members is done through the website, not the npm CLI.

To use teams to manage permissions on packages belonging to your organization, use the **npm access** command to grant or revoke the appropriate permissions.

# See Also

- [npm-access](#)
- [npm-registry](#)

---

Last modified January 08, 2016

Found a typo? Send a [pull request!](#)

## npm-test

# Synopsis

```
npm test [-- <args>]
npm tst [-- <args>]
```

# Description

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

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

# See Also

- [npm-run-script](#)
- [npm-scripts](#)
- [npm-start](#)
- [npm-restart](#)
- [npm-stop](#)

---

Last modified January 08, 2016

Found a typo? Send a [pull request!](#)

## npm-uninstall

# Synopsis

```
npm uninstall [<@scope>/]<pkg>[@<version>]... [-S|--save|-D|--save-dev|-O|--save-optional]
aliases: remove, rm, r, un, unlink
```

# Description

This uninstalls a package, completely removing everything npm installed on its behalf.

Example:

```
npm uninstall sax
```

In global mode (ie, with `-g` or `--global` appended to the command), it uninstalls the current package context as a global package.

`npm uninstall` takes 3 exclusive, optional flags which save or update the package version in your main package.json:

- **-S, --save** : Package will be removed from your **dependencies** .
- **-D, --save-dev** : Package will be removed from your **devDependencies** .
- **-O, --save-optional** : Package will be removed from your **optionalDependencies** .

Further, if you have an **npm-shrinkwrap.json** then it will be updated as well.

Scope is optional and follows the usual rules for [npm-scope](#) .

Examples:

```
npm uninstall sax --save
npm uninstall @myorg/privatepackage --save
npm uninstall node-tap --save-dev
npm uninstall dtrace-provider --save-optional
```

## See Also

- [npm-prune](#)
- [npm-install](#)
- [npm-folders](#)
- [npm-config](#)
- [npm-config](#)
- [npmrc](#)

---

Last modified January 08, 2016      Found a typo? Send a [pull request!](#)

## npm-unpublish

## Synopsis

```
npm unpublish [<@scope>/]<pkg>[@<version>]
```

## WARNING

It is generally considered bad behavior to remove versions of a library that others are depending on!

Consider using the **deprecate** command instead, if your intent is to encourage users to upgrade.

There is plenty of room on the registry.

## Description

This removes a package version from the registry, deleting its entry and removing the tarball.

If no version is specified, or if all versions are removed then the root package entry is removed from the registry entirely.

Even if a package version is unpublished, that specific name and version combination can never be reused. In order to publish the package again, a new version number must be used.

The scope is optional and follows the usual rules for [npm-scope](#) .



## See Also

- [npm-deprecate](#)
- [npm-publish](#)
- [npm-registry](#)
- [npm-adduser](#)
- [npm-owner](#)

---

Last modified January 08, 2016

Found a typo? Send a [pull request!](#)

## npm-update

## Synopsis

```
npm update [-g] [<pkg>...]
```

## Description

This command will update all the packages listed to the latest version (specified by the `tag` config), respecting semver.

It will also install missing packages. As with all commands that install packages, the `--dev` flag will cause `devDependencies` to be processed as well.

If the `-g` flag is specified, this command will update globally installed packages.

If no package name is specified, all packages in the specified location (global or local) will be updated.

As of **npm@2.6.1**, the `npm update` will only inspect top-level packages. Prior versions of `npm` would also recursively inspect all dependencies. To get the old behavior, use `npm --depth Infinity update`, but be warned that simultaneous asynchronous update of all packages, including `npm` itself and packages that `npm` depends on, often causes problems up to and including the uninstallation of `npm` itself.

To restore a missing `npm`, use the command:

```
curl -L https://npmjs.com/install.sh | sh
```

## EXAMPLES

IMPORTANT VERSION NOTE: these examples assume **npm@2.6.1** or later. For older versions of `npm`, you must specify `--depth 0` to get the behavior described below.

For the examples below, assume that the current package is `app` and it depends on dependencies, `dep1` ( `dep2` , .. etc.). The published versions of `dep1` are:

```
{
  "dist-tags": { "latest": "1.2.2" },
  "versions": {
    "1.2.2",
    "1.2.1",
    "1.2.0",
    "1.1.2",
    "1.1.1",
    "1.0.0",
    "0.4.1",
    "0.4.0",
    "0.2.0"
  }
}
```

### Caret Dependencies

If `app` 's `package.json` contains:

```
"dependencies": {
  "dep1": "^1.1.1"
}
```

Then `npm update` will install `dep1@1.2.2`, because `1.2.2` is `latest` and `1.2.2` satisfies `^1.1.1`.

### Tilde Dependencies

However, if `app` 's `package.json` contains:

```
"dependencies": {
  "dep1": "~1.1.1"
}
```

In this case, running `npm update` will install `dep1@1.1.2`. Even though the `latest` tag points to `1.2.2`, this version does not satisfy `~1.1.1`, which is equivalent to `>=1.1.1 <1.2.0`. So the highest-sorting version that satisfies `~1.1.1` is used, which is `1.1.2`.

### Caret Dependencies below 1.0.0

Suppose `app` has a caret dependency on a version below `1.0.0`, for example:

```
"dependencies": {
  "dep1": "^0.2.0"
}
```

`npm update` will install `dep1@0.2.0`, because there are no other versions which satisfy `^0.2.0`.

If the dependence were on `^0.4.0`:

```
"dependencies": {
  "dep1": "^0.4.0"
}
```

Then `npm update` will install `dep1@0.4.1`, because that is the highest-sorting version that satisfies `^0.4.0` (`>= 0.4.0 <0.5.0`).

### Recording Updates with `--save`

When you want to update a package and save the new version as the minimum required dependency in `package.json`, you can use `npm update -S` or `npm update --save`. For example if `package.json` contains:

```
"dependencies": {
  "dep1": "^1.1.1"
}
```

Then `npm update --save` will install `dep1@1.2.2` (i.e., `latest`), and `package.json` will be modified:

```
"dependencies": {
  "dep1": "^1.2.2"
}
```

Note that `npm` will only write an updated version to `package.json` if it installs a new package.

### Updating Globally-Installed Packages

`npm update -g` will apply the `update` action to each globally installed package that is `outdated` -- that is, has a version that is different from `latest`.

NOTE: If a package has been upgraded to a version newer than `latest`, it will be *downgraded*.

## See Also

- [npm-install](#)
- [npm-outdated](#)
- [npm-shrinkwrap](#)
- [npm-registry](#)
- [npm-folders](#)
- [npm-ls](#)

## npm-version

# Synopsis

```
npm version [<newversion> | major | minor | patch | premajor | preminor | prepatch | prerelease | from-git]
```

```
'npm [-v | --version]' to print npm version
'npm view <pkg> version' to view a package's published version
'npm ls' to inspect current package/dependency versions
```

# Description

Run this in a package directory to bump the version and write the new data back to **package.json** and, if present, **npm-shrinkwrap.json**.

The **newversion** argument should be a valid semver string, a valid second argument to [semver.inc](#) (one of **patch**, **minor**, **major**, **prepatch**, **preminor**, **premajor**, **prerelease**), or **from-git**. In the second case, the existing version will be incremented by 1 in the specified field. **from-git** will try to read the latest git tag, and use that as the new npm version.

If run in a git repo, it will also create a version commit and tag. This behavior is controlled by **git-tag-version** (see below), and can be disabled on the command line by running **npm --no-git-tag-version version**. It will fail if the working directory is not clean, unless the **-f** or **--force** flag is set.

If supplied with **-m** or **--message** config option, npm will use it as a commit message when creating a version commit. If the **message** config contains **%s** then that will be replaced with the resulting version number. For example:

```
npm version patch -m "Upgrade to %s for reasons"
```

If the **sign-git-tag** config is set, then the tag will be signed using the **-s** flag to git. Note that you must have a default GPG key set up in your git config for this to work properly. For example:

```
$ npm config set sign-git-tag true
$ npm version patch
```

```
You need a passphrase to unlock the secret key for
user: "isaacs (http://blog.izs.me/) <i@izs.me>"
2048-bit RSA key, ID 6C481CF6, created 2010-08-31
```

```
Enter passphrase:
```

If **preversion**, **version**, or **postversion** are in the **scripts** property of the package.json, they will be executed as part of running **npm version**.

The exact order of execution is as follows:

1. Check to make sure the git working directory is clean before we get started. Your scripts may add files to the commit in future steps. This step is skipped if the **--force** flag is set.
2. Run the **preversion** script. These scripts have access to the old **version** in package.json. A typical use would be running your full test suite before deploying. Any files you want added to the commit should be explicitly added using **git add**.
3. Bump **version** in **package.json** as requested ( **patch**, **minor**, **major**, etc).
4. Run the **version** script. These scripts have access to the new **version** in package.json (so they can incorporate it into file headers in generated files for example). Again, scripts should explicitly add generated files to the commit using **git add**.
5. Commit and tag.
6. Run the **postversion** script. Use it to clean up the file system or automatically push the commit and/or tag.

Take the following example:

```
"scripts": {
  "preversion": "npm test",
  "version": "npm run build && git add -A dist",
  "postversion": "git push && git push --tags && rm -rf build/temp"
}
```

This runs all your tests, and proceeds only if they pass. Then runs your **build** script, and adds everything in the **dist** directory to the commit. After the commit, it pushes the new commit and tag up to the server, and deletes the **build/temp** directory.

## Configuration

### git-tag-version

- Default: true
- Type: Boolean

Commit and tag the version change.

## See Also

- [npm-init](#)
- [npm-run-script](#)
- [npm-scripts](#)
- [package.json](#)
- [semver](#)
- [config](#)

---

Last modified January 21, 2016      Found a typo? Send a [pull request!](#)

## npm-view

## Synopsis

```
npm view [<@scope>/]<name>[@<version>] [<field>[.<subfield>]...]  
aliases: info, show, v
```

## Description

This command shows data about a package and prints it to the stream referenced by the **outfd** config, which defaults to stdout.

To show the package registry entry for the **connect** package, you can do this:

```
npm view connect
```

The default version is "latest" if unspecified.

Field names can be specified after the package descriptor. For example, to show the dependencies of the **ronn** package at version 0.3.5, you could do the following:

```
npm view ronn@0.3.5 dependencies
```

You can view child fields by separating them with a period. To view the git repository URL for the latest version of npm, you could do this:

```
npm view npm repository.url
```

This makes it easy to view information about a dependency with a bit of shell scripting. For example, to view all the data about the version of opts that ronn depends on, you can do this:

```
npm view opts@$(npm view ronn dependencies.opts)
```

For fields that are arrays, requesting a non-numeric field will return all of the values from the objects in the list. For example, to get all the contributor names for the "express" project, you can do this:

```
npm view express contributors.email
```

You may also use numeric indices in square braces to specifically select an item in an array field. To just get the email address of the first contributor in the list, you can do this:

```
npm view express contributors[0].email
```

Multiple fields may be specified, and will be printed one after another. For example, to get all the contributor names and email addresses, you can do this:

```
npm view express contributors.name contributors.email
```

"Person" fields are shown as a string if they would be shown as an object. So, for example, this will show the list of npm contributors in the shortened string format. (See [package.json](#) for more on this.)

```
npm view npm contributors
```

If a version range is provided, then data will be printed for every matching version of the package. This will show which version of jsdom was required by each matching version of yui3:

```
npm view yui3@'>0.5.4' dependencies.jsdom
```

To show the **connect** package version history, you can do this:

```
npm view connect versions
```

## OUTPUT

If only a single string field for a single version is output, then it will not be colorized or quoted, so as to enable piping the output to another command. If the field is an object, it will be output as a JavaScript object literal.

If the `--json` flag is given, the outputted fields will be JSON.

If the version range matches multiple versions, then each printed value will be prefixed with the version it applies to.

If multiple fields are requested, then each of them are prefixed with the field name.

## See Also

- [npm-search](#)
- [npm-registry](#)
- [npm-config](#)
- [npm-config](#)
- [npmrc](#)
- [npm-docs](#)

# Synopsis

```
npm whoami [--registry <registry>]
```

## Description

Print the `username` config to standard output.

## See Also

- [npm-config](#)
- [npm-config](#)
- [npmrc](#)
- [npm-adduser](#)

---

Last modified January 08, 2016    Found a typo? Send a [pull request!](#)

## npm-folders

## Description

npm puts various things on your computer. That's its job.

This document will tell you what it puts where.

tl;dr

- Local install (default): puts stuff in `./node_modules` of the current package root.
- Global install (with `-g`): puts stuff in `/usr/local` or wherever node is installed.
- Install it **locally** if you're going to `require()` it.
- Install it **globally** if you're going to run it on the command line.
- If you need both, then install it in both places, or use `npm link`.

### prefix Configuration

The `prefix` config defaults to the location where node is installed. On most systems, this is `/usr/local`. On windows, this is the exact location of the `node.exe` binary. On Unix systems, it's one level up, since node is typically installed at `{prefix}/bin/node` rather than `{prefix}/node.exe`.

When the `global` flag is set, npm installs things into this prefix. When it is not set, it uses the root of the current package, or the current working directory if not in a package already.

### Node Modules

Packages are dropped into the `node_modules` folder under the `prefix`. When installing locally, this means that you can `require("packagename")` to load its main module, or `require("packagename/lib/path/to/sub/module")` to load other modules.

Global installs on Unix systems go to `{prefix}/lib/node_modules`. Global installs on Windows go to `{prefix}/node_modules` (that is, no `lib` folder.)

Scoped packages are installed the same way, except they are grouped together in a sub-folder of the relevant `node_modules` folder with the name of that scope prefix by the `@` symbol, e.g. `npm install @myorg/package` would place the package in `{prefix}/node_modules/@myorg/package`. See [scope](#) for more details.

If you wish to `require()` a package, then install it locally.

## Executables

When in global mode, executables are linked into `{prefix}/bin` on Unix, or directly into `{prefix}` on Windows.

When in local mode, executables are linked into `./node_modules/.bin` so that they can be made available to scripts run through npm. (For example, so that a test runner will be in the path when you run `npm test`.)

## Man Pages

When in global mode, man pages are linked into `{prefix}/share/man`.

When in local mode, man pages are not installed.

Man pages are not installed on Windows systems.

## Cache

See [npm-cache](#). Cache files are stored in `~/.npm` on Posix, or `~/npm-cache` on Windows.

This is controlled by the `cache` configuration param.

## Temp Files

Temporary files are stored by default in the folder specified by the `tmp` config, which defaults to the TMPDIR, TMP, or TEMP environment variables, or `/tmp` on Unix and `c:\windows\temp` on Windows.

Temp files are given a unique folder under this root for each run of the program, and are deleted upon successful exit.

# More Information

When installing locally, npm first tries to find an appropriate `prefix` folder. This is so that `npm install foo@1.2.3` will install to the sensible root of your package, even if you happen to have `cd` ed into some other folder.

Starting at the `$PWD`, npm will walk up the folder tree checking for a folder that contains either a `package.json` file, or a `node_modules` folder. If such a thing is found, then that is treated as the effective "current directory" for the purpose of running npm commands. (This behavior is inspired by and similar to git's .git-folder seeking logic when running git commands in a working dir.)

If no package root is found, then the current folder is used.

When you run `npm install foo@1.2.3`, then the package is loaded into the cache, and then unpacked into `./node_modules/foo`. Then, any of foo's dependencies are similarly unpacked into `./node_modules/foo/node_modules/...`.

Any bin files are symlinked to `./node_modules/.bin/`, so that they may be found by npm scripts when necessary.

## Global Installation

If the `global` configuration is set to true, then npm will install packages "globally".

For global installation, packages are installed roughly the same way, but using the folders described above.

## Cycles, Conflicts, and Folder Parsimony

Cycles are handled using the property of node's module system that it walks up the directories looking for `node_modules` folders. So, at every stage, if a package is already installed in an ancestor `node_modules` folder, then it is not installed at the current location.

Consider the case above, where `foo -> bar -> baz`. Imagine if, in addition to that, baz depended on bar, so you'd have: `foo -> bar -> baz -> bar -> baz ...`. However, since the folder structure is: `foo/node_modules/bar/node_modules/baz`, there's no need to put another copy of bar into `.../baz/node_modules`, since when it calls `require("bar")`, it will get the copy that is installed in `foo/node_modules/bar`.

This shortcut is only used if the exact same version would be installed in multiple nested `node_modules` folders. It is still possible to have `a/node_modules/b/node_modules/a` if the two "a" packages are different versions. However, without repeating the exact same package multiple times, an infinite regress will always be prevented.

Another optimization can be made by installing dependencies at the highest level possible, below the localized "target" folder.

## Example

Consider this dependency graph:

```

foo
+-- blerg@1.2.5
+-- bar@1.2.3
|   +-- blerg@1.x (latest=1.3.7)
|   +-- baz@2.x
|   |   `-- quux@3.x
|   |       `-- bar@1.2.3 (cycle)
|   `-- asdf@*
|-- baz@1.2.3
|   `-- quux@3.x
|       `-- bar

```

In this case, we might expect a folder structure like this:

```

foo
+-- node_modules
|   +-- blerg (1.2.5) <---[A]
|   +-- bar (1.2.3) <---[B]
|   |   `-- node_modules
|   |       +-- baz (2.0.2) <---[C]
|   |       |   `-- node_modules
|   |       |       `-- quux (3.2.0)
|   |       `-- asdf (2.3.4)
|-- baz (1.2.3) <---[D]
|   `-- node_modules
|       `-- quux (3.2.0) <---[E]

```

Since foo depends directly on **bar@1.2.3** and **baz@1.2.3**, those are installed in foo's **node\_modules** folder.

Even though the latest copy of blerg is 1.3.7, foo has a specific dependency on version 1.2.5. So, that gets installed at [A]. Since the parent installation of blerg satisfies bar's dependency on **blerg@1.x**, it does not install another copy under [B].

Bar [B] also has dependencies on baz and asdf, so those are installed in bar's **node\_modules** folder. Because it depends on **baz@2.x**, it cannot re-use the **baz@1.2.3** installed in the parent **node\_modules** folder [D], and must install its own copy [C].

Underneath bar, the **baz -> quux -> bar** dependency creates a cycle. However, because bar is already in quux's ancestry [B], it does not unpack another copy of bar into that folder.

Underneath **foo -> baz** [D], quux's [E] folder tree is empty, because its dependency on bar is satisfied by the parent folder copy installed at [B].

For a graphical breakdown of what is installed where, use **npm ls**.

## Publishing

Upon publishing, npm will look in the **node\_modules** folder. If any of the items there are not in the **bundledDependencies** array, then they will not be included in the package tarball.

This allows a package maintainer to install all of their dependencies (and dev dependencies) locally, but only re-publish those items that cannot be found elsewhere. See [package.json](#) for more information.

## See Also

- [npm-faq](#)
- [package.json](#)
- [npm-install](#)
- [npm-pack](#)
- [npm-cache](#)
- [npm-config](#)
- [npmrc](#)
- [npm-config](#)
- [npm-publish](#)



## npmrc

# Description

npm gets its config settings from the command line, environment variables, and **npmrc** files.

The **npm config** command can be used to update and edit the contents of the user and global npmrc files.

For a list of available configuration options, see [npm-config](#).

## FILES

The four relevant files are:

- per-project config file (/path/to/my/project/.npmrc)
- per-user config file (~/.npmrc)
- global config file (\$PREFIX/etc/npmrc)
- npm builtin config file (/path/to/npm/npmrc)

All npm config files are an ini-formatted list of **key = value** parameters. Environment variables can be replaced using **\${VARIABLE\_NAME}** . For example:

```
prefix = ${HOME}/.npm-packages
```

Each of these files is loaded, and config options are resolved in priority order. For example, a setting in the userconfig file would override the setting in the globalconfig file.

Array values are specified by adding "[]" after the key name. For example:

```
key[] = "first value"
key[] = "second value"
```

**NOTE:** Because local (per-project or per-user) **.npmrc** files can contain sensitive credentials, they must be readable and writable *only* by your user account (i.e. must have a mode of **0600** ), otherwise they *will be ignored by npm!*

### Per-project config file

When working locally in a project, a **.npmrc** file in the root of the project (ie, a sibling of **node\_modules** and **package.json** ) will set config values specific to this project.

Note that this only applies to the root of the project that you're running npm in. It has no effect when your module is published. For example, you can't publish a module that forces itself to install globally, or in a different location.

Additionally, this file is not read in global mode, such as when running **npm install -g** .

### Per-user config file

**\$HOME/.npmrc** (or the **userconfig** param, if set in the environment or on the command line)

### Global config file

**\$PREFIX/etc/npmrc** (or the **globalconfig** param, if set above): This file is an ini-file formatted list of **key = value** parameters. Environment variables can be replaced as above.

### Built-in config file

**path/to/npm/itself/npmrc**

This is an unchangeable "builtin" configuration file that npm keeps consistent across updates. Set fields in here using the **./configure** script that comes with npm. This is primarily for distribution maintainers to override default configs in a standard and consistent manner.

## See Also

- [npm-folders](#)
- [npm-config](#)
- [npm-config](#)
- [package.json](#)
- [npm](#)

---

Last modified January 08, 2016

Found a typo? Send a [pull request!](#)

## package.json

# Description

This document is all you need to know about what's required in your package.json file. It must be actual JSON, not just a JavaScript object literal.

A lot of the behavior described in this document is affected by the config settings described in [npm-config](#) .

## name

The *most* important things in your package.json are the name and version fields. Those are actually required, and your package won't install without them. The name and version together form an identifier that is assumed to be completely unique. Changes to the package should come along with changes to the version.

The name is what your thing is called.

Some rules:

- The name must be less than or equal to 214 characters. This includes the scope for scoped packages.
- The name can't start with a dot or an underscore.
- New packages must not have uppercase letters in the name.
- The name ends up being part of a URL, an argument on the command line, and a folder name. Therefore, the name can't contain any non-URL-safe characters.

Some tips:

- Don't use the same name as a core Node module.
- Don't put "js" or "node" in the name. It's assumed that it's js, since you're writing a package.json file, and you can specify the engine using the "engines" field. (See below.)
- The name will probably be passed as an argument to require(), so it should be something short, but also reasonably descriptive.
- You may want to check the npm registry to see if there's something by that name already, before you get too attached to it. <https://www.npmjs.com/>

A name can be optionally prefixed by a scope, e.g. `@myorg/mypackage` . See [npm-scope](#) for more detail.

## version

The *most* important things in your package.json are the name and version fields. Those are actually required, and your package won't install without them. The name and version together form an identifier that is assumed to be completely unique. Changes to the package should come along with changes to the version.

Version must be parseable by [node-semver](#), which is bundled with npm as a dependency. ( `npm install semver` to use it yourself.)

More on version numbers and ranges at [semver](#).

## description

Put a description in it. It's a string. This helps people discover your package, as it's listed in `npm search` .

# keywords

Put keywords in it. It's an array of strings. This helps people discover your package as it's listed in `npm search`.

# homepage

The url to the project homepage.

**NOTE:** This is *not* the same as "url". If you put a "url" field, then the registry will think it's a redirection to your package that has been published somewhere else, and spit at you.

Literally. Spit. I'm so not kidding.

# bugs

The url to your project's issue tracker and / or the email address to which issues should be reported. These are helpful for people who encounter issues with your package.

It should look like this:

```
{ "url" : "https://github.com/owner/project/issues"
, "email" : "project@hostname.com"
}
```

You can specify either one or both values. If you want to provide only a url, you can specify the value for "bugs" as a simple string instead of an object.

If a url is provided, it will be used by the `npm bugs` command.

# license

You should specify a license for your package so that people know how they are permitted to use it, and any restrictions you're placing on it.

If you're using a common license such as BSD-2-Clause or MIT, add a current SPDX license identifier for the license you're using, like this:

```
{ "license" : "BSD-3-Clause" }
```

You can check [the full list of SPDX license IDs](#). Ideally you should pick one that is [OSI](#) approved.

If your package is licensed under multiple common licenses, use an [SPDX license expression syntax version 2.0 string](#), like this:

```
{ "license" : "(ISC OR GPL-3.0)" }
```

If you are using a license that hasn't been assigned an SPDX identifier, or if you are using a custom license, use a string value like this one:

```
{ "license" : "SEE LICENSE IN <filename>" }
```

Then include a file named `<filename>` at the top level of the package.

Some old packages used license objects or a "licenses" property containing an array of license objects:

```
// Not valid metadata
{ "license" :
  { "type" : "ISC"
  , "url" : "http://opensource.org/licenses/ISC"
  }
}

// Not valid metadata
{ "licenses" :
  [
    { "type": "MIT"
    }
```

```

    , "url": "http://www.opensource.org/licenses/mit-license.php"
  }
  , { "type": "Apache-2.0"
    , "url": "http://opensource.org/licenses/apache2.0.php"
    }
  ]
}

```

Those styles are now deprecated. Instead, use SPDX expressions, like this:

```

{ "license": "ISC" }

{ "license": "(MIT OR Apache-2.0)" }

```

Finally, if you do not wish to grant others the right to use a private or unpublished package under any terms:

```
{ "license": "UNLICENSED" }
```

Consider also setting **"private": true** to prevent accidental publication.

## people fields: author, contributors

The "author" is one person. "contributors" is an array of people. A "person" is an object with a "name" field and optionally "url" and "email", like this:

```

{ "name" : "Barney Rubble"
  , "email" : "b@rubble.com"
  , "url" : "http://barnyrubble.tumblr.com/"
}

```

Or you can shorten that all into a single string, and npm will parse it for you:

```
"Barney Rubble <b@rubble.com> (http://barnyrubble.tumblr.com/)"
```

Both email and url are optional either way.

npm also sets a top-level "maintainers" field with your npm user info.

## files

The "files" field is an array of files to include in your project. If you name a folder in the array, then it will also include the files inside that folder. (Unless they would be ignored by another rule.)

You can also provide a ".npmignore" file in the root of your package or in subdirectories, which will keep files from being included, even if they would be picked up by the files array. The **.npmignore** file works just like a **.gitignore**.

Certain files are always included, regardless of settings:

- **package.json**
- **README** (and its variants)
- **CHANGELOG** (and its variants)
- **LICENSE / LICENCE**

Conversely, some files are always ignored:

- **.git**
- **CVS**
- **.svn**
- **.hg**
- **.lock-wscript**
- **.wafpickle-N**
- **\*.swp**
- **.DS\_Store**

- `._*`
- `npm-debug.log`

## main

The `main` field is a module ID that is the primary entry point to your program. That is, if your package is named `foo`, and a user installs it, and then does `require("foo")`, then your main module's exports object will be returned.

This should be a module ID relative to the root of your package folder.

For most modules, it makes the most sense to have a main script and often not much else.

## bin

A lot of packages have one or more executable files that they'd like to install into the `PATH`. npm makes this pretty easy (in fact, it uses this feature to install the "npm" executable.)

To use this, supply a `bin` field in your package.json which is a map of command name to local file name. On install, npm will symlink that file into `prefix/bin` for global installs, or `./node_modules/.bin/` for local installs.

For example, myapp could have this:

```
{ "bin" : { "myapp" : "./cli.js" } }
```

So, when you install myapp, it'll create a symlink from the `cli.js` script to `/usr/local/bin/myapp`.

If you have a single executable, and its name should be the name of the package, then you can just supply it as a string. For example:

```
{ "name": "my-program"
, "version": "1.2.5"
, "bin": "./path/to/program" }
```

would be the same as this:

```
{ "name": "my-program"
, "version": "1.2.5"
, "bin" : { "my-program" : "./path/to/program" } }
```

## man

Specify either a single file or an array of filenames to put in place for the `man` program to find.

If only a single file is provided, then it's installed such that it is the result from `man <pkgname>`, regardless of its actual filename. For example:

```
{ "name" : "foo"
, "version" : "1.2.3"
, "description" : "A packaged foo footer for fooing foos"
, "main" : "foo.js"
, "man" : "./man/doc.1"
}
```

would link the `./man/doc.1` file in such that it is the target for `man foo`

If the filename doesn't start with the package name, then it's prefixed. So, this:

```
{ "name" : "foo"
, "version" : "1.2.3"
, "description" : "A packaged foo footer for fooing foos"
, "main" : "foo.js"
, "man" : [ "./man/foo.1", "./man/bar.1" ]
}
```

will create files to do `man foo` and `man foo-bar` .

Man files must end with a number, and optionally a `.gz` suffix if they are compressed. The number dictates which man section the file is installed into.

```
{ "name" : "foo"
, "version" : "1.2.3"
, "description" : "A packaged foo footer for footing foos"
, "main" : "foo.js"
, "man" : [ "./man/foo.1", "./man/foo.2" ]
}
```

will create entries for `man foo` and `man 2 foo`

## directories

The CommonJS [Packages](#) spec details a few ways that you can indicate the structure of your package using a `directories` object. If you look at [npm's package.json](#), you'll see that it has directories for doc, lib, and man.

In the future, this information may be used in other creative ways.

### directories.lib

Tell people where the bulk of your library is. Nothing special is done with the lib folder in any way, but it's useful meta info.

### directories.bin

If you specify a `bin` directory in `directories.bin` , all the files in that folder will be added.

Because of the way the `bin` directive works, specifying both a `bin` path and setting `directories.bin` is an error. If you want to specify individual files, use `bin` , and for all the files in an existing `bin` directory, use `directories.bin` .

### directories.man

A folder that is full of man pages. Sugar to generate a "man" array by walking the folder.

### directories.doc

Put markdown files in here. Eventually, these will be displayed nicely, maybe, someday.

### directories.example

Put example scripts in here. Someday, it might be exposed in some clever way.

## repository

Specify the place where your code lives. This is helpful for people who want to contribute. If the git repo is on GitHub, then the `npm docs` command will be able to find you.

Do it like this:

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

"repository" :
{ "type" : "svn"
, "url" : "https://v8.googlecode.com/svn/trunk/"
}
```

The URL should be a publicly available (perhaps read-only) url that can be handed directly to a VCS program without any modification. It should not be a url to an html project page that you put in your browser. It's for computers.

For GitHub, GitHub gist, Bitbucket, or GitLab repositories you can use the same shortcut syntax you use for `npm install` :

```
"repository": "npm/npm"

"repository": "gist:11081aaa281"

"repository": "bitbucket:example/repo"

"repository": "gitlab:another/repo"
```

## scripts

The "scripts" property is a dictionary containing script commands that are run at various times in the lifecycle of your package. The key is the lifecycle event, and the value is the command to run at that point.

See [npm-scripts](#) to find out more about writing package scripts.

## config

A "config" object can be used to set configuration parameters used in package scripts that persist across upgrades. For instance, if a package had the following:

```
{ "name" : "foo"
  , "config" : { "port" : "8080" } }
```

and then had a "start" command that then referenced the `npm_package_config_port` environment variable, then the user could override that by doing `npm config set foo:port 8001`.

See [npm-config](#) and [npm-scripts](#) for more on package configs.

## dependencies

Dependencies are specified in a simple object that maps a package name to a version range. The version range is a string which has one or more space-separated descriptors. Dependencies can also be identified with a tarball or git URL.

Please do not put test harnesses or transpilers in your `dependencies` object. See `devDependencies`, below.

See [semver](#) for more details about specifying version ranges.

- **version** Must match **version** exactly
- **>version** Must be greater than **version**
- **>=version** etc
- **<version**
- **<=version**
- **~version** "Approximately equivalent to version" See [semver](#)
- **^version** "Compatible with version" See [semver](#)
- **1.2.x** 1.2.0, 1.2.1, etc., but not 1.3.0
- **http://...** See 'URLs as Dependencies' below
- **\*** Matches any version
- **""** (just an empty string) Same as **\***
- **version1 - version2** Same as **>=version1 <=version2**.
- **range1 || range2** Passes if either range1 or range2 are satisfied.
- **git...** See 'Git URLs as Dependencies' below
- **user/repo** See 'GitHub URLs' below
- **tag** A specific version tagged and published as **tag** See [npm-tag](#)
- **path/path/path** See [Local Paths](#) below

For example, these are all valid:

```
{ "dependencies" :
  { "foo" : "1.0.0 - 2.9999.9999"
    , "bar" : ">=1.0.2 <2.1.2"
    , "baz" : ">1.0.2 <=2.3.4"
```

```

, "boo" : "2.0.1"
, "qux" : "<1.0.0 || >=2.3.1 <2.4.5 || >=2.5.2 <3.0.0"
, "asd" : "http://asdf.com/asdf.tar.gz"
, "til" : "~1.2"
, "elf" : "~1.2.3"
, "two" : "2.x"
, "thr" : "3.3.x"
, "lat" : "latest"
, "dyl" : "file:../dyl"
}
}

```

## URLs as Dependencies

You may specify a tarball URL in place of a version range.

This tarball will be downloaded and installed locally to your package at install time.

## Git URLs as Dependencies

Git urls can be of the form:

```

git://github.com/user/project.git#commit-ish
git+ssh://user@hostname:project.git#commit-ish
git+ssh://user@hostname/project.git#commit-ish
git+http://user@hostname/project/blah.git#commit-ish
git+https://user@hostname/project/blah.git#commit-ish

```

The **commit-ish** can be any tag, sha, or branch which can be supplied as an argument to **git checkout**. The default is **master**.

# GitHub URLs

As of version 1.1.65, you can refer to GitHub urls as just "foo": "user/foo-project". Just as with git URLs, a **commit-ish** suffix can be included. For example:

```

{
  "name": "foo",
  "version": "0.0.0",
  "dependencies": {
    "express": "visionmedia/express",
    "mocha": "visionmedia/mocha#4727d357ea"
  }
}

```

# Local Paths

As of version 2.0.0 you can provide a path to a local directory that contains a package. Local paths can be saved using **npm install -S** or **npm install --save**, using any of these forms:

```

../foo/bar
~/foo/bar
./foo/bar
/foo/bar

```

in which case they will be normalized to a relative path and added to your **package.json**. For example:

```

{
  "name": "baz",
  "dependencies": {
    "bar": "file:../foo/bar"
  }
}

```

This feature is helpful for local offline development and creating tests that require npm installing where you don't want to hit an external server, but should not be used when publishing packages to the public registry.



# devDependencies

If someone is planning on downloading and using your module in their program, then they probably don't want or need to download and build the external test or documentation framework that you use.

In this case, it's best to map these additional items in a `devDependencies` object.

These things will be installed when doing `npm link` or `npm install` from the root of a package, and can be managed like any other npm configuration param. See [npm-config](#) for more on the topic.

For build steps that are not platform-specific, such as compiling CoffeeScript or other languages to JavaScript, use the `prepublish` script to do this, and make the required package a devDependency.

For example:

```
{ "name": "ethopia-waza",
  "description": "a delightfully fruity coffee varietal",
  "version": "1.2.3",
  "devDependencies": {
    "coffee-script": "~1.6.3"
  },
  "scripts": {
    "prepublish": "coffee -o lib/ -c src/waza.coffee"
  },
  "main": "lib/waza.js"
}
```

The `prepublish` script will be run before publishing, so that users can consume the functionality without requiring them to compile it themselves. In dev mode (ie, locally running `npm install`), it'll run this script as well, so that you can test it easily.

# peerDependencies

In some cases, you want to express the compatibility of your package with a host tool or library, while not necessarily doing a `require` of this host. This is usually referred to as a *plugin*. Notably, your module may be exposing a specific interface, expected and specified by the host documentation.

For example:

```
{
  "name": "tea-latte",
  "version": "1.3.5",
  "peerDependencies": {
    "tea": "2.x"
  }
}
```

This ensures your package `tea-latte` can be installed *along* with the second major version of the host package `tea` only. `npm install tea-latte` could possibly yield the following dependency graph:

```
└─ tea-latte@1.3.5
   └─ tea@2.2.0
```

**NOTE:** npm versions 1 and 2 will automatically install `peerDependencies` if they are not explicitly depended upon higher in the dependency tree. In the next major version of npm (npm@3), this will no longer be the case. You will receive a warning that the peerDependency is not installed instead. The behavior in npms 1 & 2 was frequently confusing and could easily put you into dependency hell, a situation that npm is designed to avoid as much as possible.

Trying to install another plugin with a conflicting requirement will cause an error. For this reason, make sure your plugin requirement is as broad as possible, and not to lock it down to specific patch versions.

Assuming the host complies with [semver](#), only changes in the host package's major version will break your plugin. Thus, if you've worked with every 1.x version of the host package, use `"^1.0"` or `"1.x"` to express this. If you depend on features introduced in 1.5.2, use `">= 1.5.2 < 2"`.

# bundledDependencies

Array of package names that will be bundled when publishing the package.

If this is spelled **"bundleDependencies"** , then that is also honored.

## optionalDependencies

If a dependency can be used, but you would like npm to proceed if it cannot be found or fails to install, then you may put it in the **optionalDependencies** object. This is a map of package name to version or url, just like the **dependencies** object. The difference is that build failures do not cause installation to fail.

It is still your program's responsibility to handle the lack of the dependency. For example, something like this:

```
try {
  var foo = require('foo')
  var fooVersion = require('foo/package.json').version
} catch (er) {
  foo = null
}
if ( notGoodFooVersion(fooVersion) ) {
  foo = null
}

// .. then later in your program ..

if (foo) {
  foo.doFooThings()
}
```

Entries in **optionalDependencies** will override entries of the same name in **dependencies** , so it's usually best to only put in one place.

## engines

You can specify the version of node that your stuff works on:

```
{ "engines" : { "node" : ">=0.10.3 <0.12" } }
```

And, like with dependencies, if you don't specify the version (or if you specify "" as the version), then any version of node will do.

If you specify an "engines" field, then npm will require that "node" be somewhere on that list. If "engines" is omitted, then npm will just assume that it works on node.

You can also use the "engines" field to specify which versions of npm are capable of properly installing your program. For example:

```
{ "engines" : { "npm" : "~1.0.20" } }
```

Note that, unless the user has set the **engine-strict** config flag, this field is advisory only.

## engineStrict

This feature was deprecated with npm 3.0.0

Prior to npm 3.0.0, this feature was used to treat this package as if the user had set **engine-strict** .

## os

You can specify which operating systems your module will run on:

```
"os" : [ "darwin", "linux" ]
```

You can also blacklist instead of whitelist operating systems, just prepend the blacklisted os with a '!':

```
"os" : [ "!win32" ]
```

The host operating system is determined by `process.platform`

It is allowed to both blacklist, and whitelist, although there isn't any good reason to do this.

## cpu

If your code only runs on certain cpu architectures, you can specify which ones.

```
"cpu" : [ "x64", "ia32" ]
```

Like the `os` option, you can also blacklist architectures:

```
"cpu" : [ "!arm", "!mips" ]
```

The host architecture is determined by `process.arch`

## preferGlobal

If your package is primarily a command-line application that should be installed globally, then set this value to `true` to provide a warning if it is installed locally.

It doesn't actually prevent users from installing it locally, but it does help prevent some confusion if it doesn't work as expected.

## private

If you set `"private": true` in your package.json, then npm will refuse to publish it.

This is a way to prevent accidental publication of private repositories. If you would like to ensure that a given package is only ever published to a specific registry (for example, an internal registry), then use the `publishConfig` dictionary described below to override the `registry` config param at publish-time.

## publishConfig

This is a set of config values that will be used at publish-time. It's especially handy if you want to set the tag, registry or access, so that you can ensure that a given package is not tagged with "latest", published to the global public registry or that a scoped module is private by default.

Any config values can be overridden, but of course only "tag", "registry" and "access" probably matter for the purposes of publishing.

See [npm-config](#) to see the list of config options that can be overridden.

## DEFAULT VALUES

npm will default some values based on package contents.

- `"scripts": {"start": "node server.js"}`

If there is a `server.js` file in the root of your package, then npm will default the `start` command to `node server.js`.

- `"scripts":{"preinstall": "node-gyp rebuild"}`

If there is a `binding.gyp` file in the root of your package, npm will default the `preinstall` command to compile using node-gyp.

- `"contributors": [...]`

If there is an `AUTHORS` file in the root of your package, npm will treat each line as a `Name <email> (url)` format, where email and url are optional. Lines which start with a `#` or are blank, will be ignored.

## See Also

- [semver](#)
- [npm-init](#)
- [npm-version](#)
- [npm-config](#)
- [npm-config](#)
- [npm-help](#)
- [npm-faq](#)
- [npm-install](#)
- [npm-publish](#)
- [npm-uninstall](#)

---

Last modified January 29, 2016

Found a typo? Send a [pull request!](#)

## npm Code of Conduct

npm exists to facilitate sharing code, by making it easy for JavaScript module developers to publish and distribute packages.

npm is a piece of technology, but more importantly, it is a community.

We believe that our mission is best served in an environment that is friendly, safe, and accepting; free from intimidation or harassment.

Towards this end, certain behaviors and practices will not be tolerated.

## tl;dr

- Be respectful.
- We're here to help: [abuse@npmjs.com](mailto:abuse@npmjs.com)
- Abusive behavior is never tolerated.
- Data published to npm is hosted at the discretion of the service administrators, and may be removed.
- Violations of this code may result in swift and permanent expulsion from the npm community.

## Scope

We expect all members of the npm community, including paid and unpaid agents, administrators, users, and customers of npm, Inc., to abide by this Code of Conduct at all times in all npm community venues, online and in person, and in one-on-one communications pertaining to npm affairs.

This policy covers the usage of the npm registry, as well as the npm website, npm related events, and any other services offered by or on behalf of npm, Inc. (collectively, the "Service"). It also applies to behavior in the context of the npm Open Source project communities, including but not limited to public GitHub repositories, IRC channels, social media, mailing lists, and public events.

This Code of Conduct is in addition to, and does not in any way nullify or invalidate, any other terms or conditions related to use of the Service.

The definitions of various subjective terms such as "discriminatory", "hateful", or "confusing" will be decided at the sole discretion of the npm abuse team.

## Friendly Harassment-Free Space

We are committed to providing a friendly, safe and welcoming environment for all, regardless of gender identity, sexual orientation, ability, ethnicity, religion, age, physical appearance, body size, race, or similar personal characteristics.

We ask that you please respect that people have differences of opinion regarding technical choices, and that every design or implementation choice carries a trade-off and numerous costs. There is seldom a single right answer. A difference of technology preferences is not a license to be rude.

Disputes over package rights must be handled respectfully, according to the terms described in the npm Dispute Resolution document. There is never a good reason to be rude over package name disputes.

Any spamming, trolling, flaming, baiting, or other attention-stealing behavior is not welcome, and will not be tolerated.

Harassing other users of the Service is never tolerated, whether via public or private media.

Avoid using offensive or harassing package names, nicknames, or other identifiers that might detract from a friendly, safe, and welcoming environment for all.

Harassment includes, but is not limited to: harmful or prejudicial verbal or written comments related to gender identity, sexual orientation, ability, ethnicity, religion, age, physical appearance, body size, race, or similar personal characteristics; inappropriate use of nudity, sexual images, and/or sexually explicit language in public spaces; threats of physical or non-physical harm; deliberate intimidation, stalking or following; harassing photography or recording; sustained disruption of talks or other events; inappropriate physical contact; and unwelcome sexual attention.

## Acceptable Package Content

The Service administrators reserve the right to make judgment calls about what is and isn't appropriate in published packages. These are guidelines to help you be successful in our community.

Packages published to the Service must be created using the npm command-line client, or a functionally equivalent implementation. For example, a "package" must not be a PNG or JPEG image, movie file, or text document. Using the Service as a personal general-purpose database is also not allowed for this reason. Packages should be npm packages, and nothing else.

Packages must contain some functionality. "Squatting", that is, publishing an empty package to "reserve" a name, is not allowed.

Packages must not contain illegal or infringing content. You should only publish packages or other materials to the Service if you have the right to do so. This includes complying with all software license agreements or other intellectual property restrictions. For example, redistributing an MIT-licensed module with the copyright notice removed, would not be allowed. You will be responsible for any violation of laws or others' intellectual property rights.

Packages must not be malware. For example, a package which is designed to maliciously exploit or damage computer systems, is not allowed. However, an explicitly documented penetration testing library designed to be used for white-hat security research would most likely be fine.

Package name, description, and other visible metadata must not include abusive, inappropriate, or harassing content.

## Reporting Violations of this Code of Conduct

If you believe someone is harassing you or has otherwise violated this Code of Conduct, please contact us at [abuse@npmjs.com](mailto:abuse@npmjs.com) to send us an abuse report. If this is the initial report of a problem, please include as much detail as possible. It is easiest for us to address issues when we have more context.

## Consequences

All content published to the Service, including user account credentials, is hosted at the sole discretion of the npm administrators.

Unacceptable behavior from any community member, including sponsors, employees, customers, or others with decision-making authority, will not be tolerated.

Anyone asked to stop unacceptable behavior is expected to comply immediately.

If a community member engages in unacceptable behavior, the npm administrators may take any action they deem appropriate, up to and including a temporary ban or permanent expulsion from the community without warning (and without refund in the case of a paid event or service).

## Addressing Grievances

If you feel you have been falsely or unfairly accused of violating this Code of Conduct, you should notify npm, Inc. We will do our best to ensure that your grievance is handled appropriately.

In general, we will choose the course of action that we judge as being most in the interest of fostering a safe and friendly community.

## Contact Info

Please contact [abuse@npmjs.com](mailto:abuse@npmjs.com) if you need to report a problem or address a grievance related to an abuse report.

You are also encouraged to contact us if you are curious about something that might be "on the line" between appropriate and inappropriate content. We are happy to provide guidance to help you be a successful part of our community.

## Changes

This is a living document and may be updated from time to time. Please refer to the [git history for this document](#) to view the changes.

## Credit and License

This Code of Conduct borrows heavily from the Stumptown Syndicate [Citizen's Code of Conduct](#), and the [Rust Project Code of Conduct](#).

This document may be reused under a [Creative Commons Attribution-ShareAlike License](#).

---

Last modified February 02, 2016      Found a typo? Send a [pull request!](#)

## Dispute Resolution

This document describes the steps that you should take to resolve module name disputes with other npm publishers.

This document is a clarification of the acceptable behavior outlined in the [npm Code of Conduct](#), and nothing in this document should be interpreted to contradict any aspect of the npm Code of Conduct.

## tl;dr

1. Get the author email with `npm owner ls <pkgname>`
2. Email the author, CC [support@npmjs.com](mailto:support@npmjs.com)
3. After a few weeks, if there's no resolution, we'll sort it out.

Don't squat on package names. Publish code or move out of the way.

## Description

There sometimes arise cases where a user publishes a module, and then later, some other user wants to use that name. Here are some common ways that happens (each of these is based on actual events.)

1. Alice writes a JavaScript module `foo`, which is not node-specific. Alice doesn't use node at all. Yusuf wants to use `foo` in node, so he wraps it in an npm module. Some time later, Alice starts using node, and wants to take over management of her program.
2. Yusuf writes an npm module `foo`, and publishes it. Perhaps much later, Alice finds a bug in `foo`, and fixes it. She sends a pull request to Yusuf, but Yusuf doesn't have the time to deal with it, because he has a new job and a new baby and is focused on his new Erlang project, and kind of not involved with node any more. Alice would like to publish a new `foo`, but can't, because the name is taken.
3. Yusuf writes a 10-line flow-control library, and calls it `foo`, and publishes it to the npm registry. Being a simple little thing, it never really has to be updated. Alice works for Foo Inc, the makers of the critically acclaimed and widely-marketed `foo` JavaScript toolkit framework. They publish it to npm as `foojs`, but people are routinely confused when `npm install foo` is some different thing.
4. Yusuf writes a parser for the widely-known `foo` file format, because he needs it for work. Then, he gets a new job, and never updates the prototype. Later on, Alice writes a much more complete `foo` parser, but can't publish, because Yusuf's `foo` is in the way.

The validity of Alice's claim in each situation can be debated. However, Alice's appropriate course of action in each case is the same.

1. `npm owner ls foo`. This will tell Alice the email address of the owner (Yusuf).
2. Alice emails Yusuf, explaining the situation **as respectfully as possible**, and what she would like to do with the module name. She adds the npm support staff [support@npmjs.com](mailto:support@npmjs.com) to the CC list of the email. Mention in the email that Yusuf can run `npm owner add alice foo` to add Alice as an owner of the `foo` package.
3. After a reasonable amount of time, if Yusuf has not responded, or if Yusuf and Alice can't come to any sort of resolution, email support [support@npmjs.com](mailto:support@npmjs.com) and we'll sort it out. ("Reasonable" is usually at least 4 weeks.)

## Reasoning

In almost every case so far, the parties involved have been able to reach an amicable resolution without any major intervention. Most people really do want to be reasonable, and are probably not even aware that they're in your way.

Module ecosystems are most vibrant and powerful when they are as self-directed as possible. If an admin one day deletes something you had worked on, then that is going to make most people quite upset, regardless of the justification. When humans solve their problems by talking to other humans with respect, everyone has the chance to end up feeling good about the interaction.

## Exceptions

Some things are not allowed, and will be removed without discussion if they are brought to the attention of the npm registry admins, including but not limited to:

1. Malware (that is, a package designed to exploit or harm the machine on which it is installed).
2. Violations of copyright or licenses (for example, cloning an MIT-licensed program, and then removing or changing the copyright and license statement).
3. Illegal content.
4. "Squatting" on a package name that you *plan* to use, but aren't actually using. Sorry, I don't care how great the name is, or how perfect a fit it is for the thing that someday might happen. If someone wants to use it today, and you're just taking up space with an empty tarball, you're going to be evicted.
5. Putting empty packages in the registry. Packages must have SOME functionality. It can be silly, but it can't be *nothing*. (See also: squatting.)
6. Doing weird things with the registry, like using it as your own personal application database or otherwise putting non-packagey things into it.
7. Other things forbidden by the npm [Code of Conduct](#) such as hateful language, pornographic content, or harassment.

If you see bad behavior like this, please report it to [abuse@npmjs.com](mailto:abuse@npmjs.com) right away. **You are never expected to resolve abusive behavior on your own.** We are here to help.

## Changes

This is a living document and may be updated from time to time. Please refer to the [git history for this document](#) to view the changes.

## License

Copyright (C) npm, Inc., All rights reserved

This document may be reused under a [Creative Commons Attribution-ShareAlike License](#).

---

Last modified February 02, 2016      Found a typo? Send a [pull request!](#)

## Copyright Policy

We take notices of potential copyright infringement seriously. If a user or other party has alleged that material on the npm website is infringing a copyright of such party or another third-party, the notice should be forwarded immediately to [abuse@npmjs.com](mailto:abuse@npmjs.com), who will work with legal counsel to resolve the dispute. If legal determines the notice satisfies all requirements under the United States Digital Millennium Copyright Act, then access to the allegedly infringing material must be promptly removed or disabled. We will then make a good faith effort to give notice of the claimed infringement to the user that posted the allegedly infringing material.

## How to Report Infringement

We respect the intellectual property of others and ask that you do too. If you believe any package or other materials available through the Service violates a copyright held by you and you would like to submit a notice pursuant to the Digital Millennium Copyright Act or other similar international law, you can submit a notice to our agent for service of notice at:

Abuse Team  
npm, Inc.  
1999 Harrison St, Ste 1150  
Oakland CA 94612 USA  
+1-510-858-7608  
[abuse@npmjs.com](mailto:abuse@npmjs.com)

Please make sure your notice meets the Digital Millennium Copyright Act requirements.

Please note that a copy of each legal notice we receive is also sent to [Lumen Database](#) for publication (with any user's personal information removed).

## How npm Responds to Notices

If the posting user objects to removal of the material, such user may file a counter notice. If we receive a counter notice from such user meeting the requirements of the DMCA, we will use good faith efforts to notify the complainant of such counter notice reinstate access to the material within 10-14 business days unless the complainant notifies us that it has filed a lawsuit against the allegedly infringing user. If we do not receive a counter notice from such user within 10 business days of giving notice of the claimed infringement, we reserve the right to permanently delete the material at issue.

We will terminate the accounts of users who are repeat infringers. Note, npm cannot provide legal advice to users (whether making a complaint or defending against a complaint), and we should ask users with legal questions to seek an attorney.

---

Last modified February 02, 2016      Found a typo? Send a [pull request!](#)

Copyright (c) npm, Inc. and Contributors All rights reserved.

npm is released under the Artistic License 2.0, subject to additional terms that are listed below.

The text of the npm License follows and the text of the additional terms follows the Artistic License 2.0 terms:

---

### The Artistic License 2.0

Copyright (c) 2000-2006, The Perl Foundation.

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

#### Preamble

This license establishes the terms under which a given free software Package may be copied, modified, distributed, and/or redistributed. The intent is that the Copyright Holder maintains some artistic control over the development of that Package while still keeping the Package available as open source and free software.

You are always permitted to make arrangements wholly outside of this license directly with the Copyright Holder of a given Package. If the terms of this license do not permit the full use that you propose to make of the Package, you should contact the Copyright Holder and seek a different licensing arrangement.

#### Definitions

"Copyright Holder" means the individual(s) or organization(s) named in the copyright notice for the entire Package.

"Contributor" means any party that has contributed code or other material to the Package, in accordance with the Copyright Holder's procedures.

"You" and "your" means any person who would like to copy, distribute, or modify the Package.

"Package" means the collection of files distributed by the Copyright Holder, and derivatives of that collection and/or of those files. A given Package may consist of either the Standard Version, or a Modified Version.

"Distribute" means providing a copy of the Package or making it accessible to anyone else, or in the case of a company or organization, to others outside of your company or organization.

"Distributor Fee" means any fee that you charge for Distributing this Package or providing support for this Package to another party. It does not mean licensing fees.

"Standard Version" refers to the Package if it has not been modified, or has been modified only in ways explicitly requested by the Copyright Holder.

"Modified Version" means the Package, if it has been changed, and such changes were not explicitly requested by the Copyright Holder.



"Original License" means this Artistic License as Distributed with the Standard Version of the Package, in its current version or as it may be modified by The Perl Foundation in the future.

"Source" form means the source code, documentation source, and configuration files for the Package.

"Compiled" form means the compiled bytecode, object code, binary, or any other form resulting from mechanical transformation or translation of the Source form.

#### Permission for Use and Modification Without Distribution

(1) You are permitted to use the Standard Version and create and use Modified Versions for any purpose without restriction, provided that you do not Distribute the Modified Version.

#### Permissions for Redistribution of the Standard Version

(2) You may Distribute verbatim copies of the Source form of the Standard Version of this Package in any medium without restriction, either gratis or for a Distributor Fee, provided that you duplicate all of the original copyright notices and associated disclaimers. At your discretion, such verbatim copies may or may not include a Compiled form of the Package.

(3) You may apply any bug fixes, portability changes, and other modifications made available from the Copyright Holder. The resulting Package will still be considered the Standard Version, and as such will be subject to the Original License.

#### Distribution of Modified Versions of the Package as Source

(4) You may Distribute your Modified Version as Source (either gratis or for a Distributor Fee, and with or without a Compiled form of the Modified Version) provided that you clearly document how it differs from the Standard Version, including, but not limited to, documenting any non-standard features, executables, or modules, and provided that you do at least ONE of the following:

(a) make the Modified Version available to the Copyright Holder of the Standard Version, under the Original License, so that the Copyright Holder may include your modifications in the Standard Version.

(b) ensure that installation of your Modified Version does not prevent the user installing or running the Standard Version. In addition, the Modified Version must bear a name that is different from the name of the Standard Version.

(c) allow anyone who receives a copy of the Modified Version to make the Source form of the Modified Version available to others under

(i) the Original License or

(ii) a license that permits the licensee to freely copy, modify and redistribute the Modified Version using the same licensing terms that apply to the copy that the licensee received, and requires that the Source form of the Modified Version, and of any works derived from it, be made freely available in that license fees are prohibited but Distributor Fees are allowed.

#### Distribution of Compiled Forms of the Standard Version or Modified Versions without the Source

(5) You may Distribute Compiled forms of the Standard Version without the Source, provided that you include complete instructions on how to get the Source of the Standard Version. Such instructions must be valid at the time of your distribution. If these instructions, at any time while you are carrying out such distribution, become invalid, you must provide new instructions on demand or cease further distribution. If you provide valid instructions or cease distribution within thirty days after you become aware that the instructions are invalid, then you do not forfeit any of your rights under this license.

(6) You may Distribute a Modified Version in Compiled form without the Source, provided that you comply with Section 4 with respect to the Source of the Modified Version.

#### Aggregating or Linking the Package

(7) You may aggregate the Package (either the Standard Version or Modified Version) with other packages and Distribute the resulting aggregation provided that you do not charge a licensing fee for the Package. Distributor Fees are permitted, and licensing fees for other components in the aggregation are permitted. The terms of this license apply to the use and Distribution of the Standard or Modified Versions as included in the aggregation.

(8) You are permitted to link Modified and Standard Versions with other works, to embed the Package in a larger work of your own, or to build stand-alone binary or bytecode versions of applications that include the Package, and Distribute the result without restriction, provided the result does not expose a direct interface to the Package.

#### Items That are Not Considered Part of a Modified Version

(9) Works (including, but not limited to, modules and scripts) that merely extend or make use of the Package, do not, by themselves, cause the Package to be a Modified Version. In addition, such works are not considered parts of the Package itself, and are not subject to the terms of this license.

#### General Provisions

(10) Any use, modification, and distribution of the Standard or Modified Versions is governed by this Artistic License. By using, modifying or distributing the Package, you accept this license. Do not use, modify, or distribute the Package, if you do not accept this license.

(11) If your Modified Version has been derived from a Modified Version made by someone other than you, you are nevertheless required to ensure that your Modified Version complies with the requirements of this license.

(12) This license does not grant you the right to use any trademark, service mark, tradename, or logo of the Copyright Holder.

(13) This license includes the non-exclusive, worldwide, free-of-charge patent license to make, have made, use, offer to sell, sell, import and otherwise transfer the Package with respect to any patent claims licensable by the Copyright Holder that are necessarily infringed by the Package. If you institute patent litigation (including a cross-claim or counterclaim) against any party alleging that the Package constitutes direct or contributory patent infringement, then this Artistic License to you shall terminate on the date that such litigation is filed.

(14) Disclaimer of Warranty: THE PACKAGE IS PROVIDED BY THE COPYRIGHT HOLDER AND CONTRIBUTORS "AS IS" AND WITHOUT ANY EXPRESS OR IMPLIED WARRANTIES. THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT ARE DISCLAIMED TO THE EXTENT PERMITTED BY YOUR LOCAL LAW. UNLESS REQUIRED BY LAW, NO COPYRIGHT HOLDER OR CONTRIBUTOR WILL BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, OR CONSEQUENTIAL DAMAGES ARISING IN ANY WAY OUT OF THE USE OF THE PACKAGE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

The following additional terms shall apply to use of the npm software, the npm website, the npm repository and any other services or products offered by npm, Inc.:

"Node.js" trademark Joyent, Inc. npm is not officially part of the Node.js project, and is neither owned by nor affiliated with Joyent, Inc.

"npm" and "The npm Registry" are owned by npm, Inc. All rights reserved.

Modules published on the npm registry are not officially endorsed by npm, Inc. or the Node.js project.

Data published to the npm registry is not part of npm itself, and is the sole property of the publisher. While every effort is made to ensure accountability, there is absolutely no guarantee, warrantee, or assertion expressed or implied as to the quality, fitness for a specific purpose, or lack of malice in any given npm package. Packages downloaded through the npm registry are independently licensed and are not covered by this license.

Additional policies relating to, and restrictions on use of, npm products and services are available on the npm website. All such policies and restrictions, as updated from time to time, are hereby incorporated into this license agreement. By using npm, you acknowledge your agreement to all such policies and restrictions.

If you have a complaint about a package in the public npm registry, and cannot resolve it with the package owner, please email [support@npmjs.com](mailto:support@npmjs.com) and explain the situation. See the [npm Dispute Resolution policy](#) for more details.

Any data published to The npm Registry (including user account information) may be removed or modified at the sole discretion of the npm server administrators.

"npm Logo" contributed by Mathias Pettersson and Brian Hammond, use is subject to <https://www.npmjs.com/policies/trademark>

"Gubblebum Blocky" font Copyright (c) by Tjarda Koster, <https://jelloween.deviantart.com> included for use in the npm website and documentation, used with permission.

This program uses several Node modules contained in the node\_modules/ subdirectory, according to the terms of their respective licenses.

Last modified February 02, 2016      Found a typo? Send a [pull request!](#)

## npm Open-Source Terms

These npm Open Source terms of use (these *Terms*) govern access to and use of <https://www.npmjs.com> (the *Website*) as well as the "npm Public Registry" at <https://registry.npmjs.org> (the *Public Registry*). Both the Website and the Public Registry are operated by npm, Inc. (*npm*). The Website and the Public Registry are referred to together as *npm Open Source* throughout these Terms.

These npm Open Source Terms were last updated on December 1, 2015. You can review prior versions at <https://github.com/npm/policies/commits/master/open-source-terms.md>.

# Important Terms

*These Terms include a number of important provisions that affect your rights and responsibilities, such as the disclaimers in "Disclaimers", limits on npm's liability to you in "Limits on Liability", and an agreement to arbitrate disputes individually in "Arbitration".*

## Other Terms

npm offers additional, paid services (*Paid Services*) that are subject to additional terms:

- Additional terms for npm Private Packages are available at <https://www.npmjs.com/policies/private-terms>.

npm Open Source and any Paid Services you may agree to use are together called *npm Services* throughout these Terms.

## Legal Agreement

You may only access or use npm Services by agreeing to these Terms. If npm adds any additional functionality to npm Services, you must agree to these Terms to use new features, too. You show your agreement with npm on these Terms by creating a user account (your *Account*) or by accessing or using npm Services without creating an account. The agreement between you and npm is a legally binding contract (this *Agreement*).

## Changes

npm may change these Terms and the additional terms for Paid Services in the future. npm will post changes on the Website with a new "last updated" date. If you have an Account, npm will notify you of changes by e-mail to the address provided for your Account, by a message on the Website, or both. If you do not have an account, npm may notify you of changes by a general announcement via the Website, but it is up to you to check for changes to these Terms. After receiving notice of changes to these Terms, you must accept those changes to continue using npm Services. You accept changes to these Terms by continuing to use npm Services. npm may change, suspend, or discontinue npm Services at any time without notice or liability to you.

## npm Policies

npm respects your privacy and limits use and sharing of information about you collected by npm Services. The privacy policy at <https://www.npmjs.com/policies/privacy> (the *Privacy Policy*) describes these policies. npm will abide by the Privacy Policy and honor the privacy settings that you choose via npm Services.

npm respects the exclusive rights of copyright holders and responds to notifications about alleged infringement via npm Services per the copyright policy at <https://www.npmjs.com/policies/dmca> (the *Copyright Policy*).

npm resolves disputes about package names in the Public Registry per the policy at <https://www.npmjs.com/policies/disputes> (*Dispute Policy*).

Use of all npm Services is governed by the code of conduct at <https://www.npmjs.com/policies/conduct> (*Code of Conduct*).

npm permits use of npm trademarks per the policy at <https://www.npmjs.com/policies/trademark>.

## Use of npm Open Source

Subject to these Terms, npm grants you permission to use npm Open Source. That permission is not exclusive to you, and you cannot transfer it to anyone else.

Your permission to use npm Open Source entitles you to do the following:

1. You may search for, download, publish, and manage packages of computer code (*Packages*) in the Public Registry, and otherwise interact with the Public Registry, via the command-line tool published by npm at <https://www.github.com/npm/npm> (the *CLI*).
2. You may search for, download, publish, and manage Packages using software other than CLI via application programming interfaces that npm publicly documents or makes available for public use (*Public APIs*).
3. You may search for and manage Packages in the Public Registry, and otherwise interact with the Public Registry, via the Website.
4. You may update and manage your Account via the Website.

# Conditions

Your permission to use npm Open Source, as well as any permission you may have to use Paid Services, are subject to the following conditions:

1. You must be at least 13 years of age to use npm Services.
2. You may not use npm Services after npm says you may not, such as by disabling your Account.
3. You must use npm Services only in accordance with "Acceptable Use".

## Acceptable Use

1. You will abide by the [Code of Conduct](#) and the [Dispute Policy](#).
2. You will not submit any content that is illegal, offensive, or otherwise harmful. You will not submit any content in violation of law, infringing the intellectual property rights of others, violating the privacy or other rights of others, or in violation of any agreement with a third party.
3. You will not disclose information that you do not have the right to disclose, such as confidential information of others.
4. You will not copy or share any personally identifiable information of any other person without their specific permission.
5. You will not violate any applicable law.
6. You will not use or attempt to use another person's Account without their specific permission.
7. You will not buy, sell, or otherwise trade in user names, organization names, names for *Packages*, or any other names reserved on *npm Services*, for money or other compensation.
8. You will not send advertisements, chain letters, or other solicitations via *npm Services*.
9. You will not automate access to, use, or monitor the Website, such as with a web crawler, browser plug-in or add-on, or other computer program that is not a web browser. You may replicate data from the Public Registry using the Public APIs per this Agreement.
10. You will not use npm Services to send e-mail to distribution lists, newsgroups, or group mail aliases.
11. You will not submit material containing malicious computer code, such as computer viruses, computer worms, rootkits, back doors, adware, or spyware, to npm Services in such a way that you should expect other users of npm Services to unwittingly execute that malicious code.
12. You will not falsely imply that you are affiliated with or endorsed by npm.
13. You will not operate illegal schemes, such as pyramid schemes, via npm Services.
14. You will not deep-hyperlink to images or other non-hypertext content served by npm Services.
15. You will not remove any marking indicating proprietary ownership from any material got via npm Services.
16. You will not display any portion of the Website via an HTML IFRAME.
17. You will not disable, avoid, or circumvent any security or access restrictions of npm Services, or access parts of npm Services not intended for access by you.
18. You will not strain infrastructure of npm Services with an unreasonable volume of requests, or requests designed to impose an unreasonable load on IT systems underlying npm Services.
19. You will not encourage or assist any other person in violation of "Acceptable Use".

## Enforcement of Acceptable Use

npm may investigate and prosecute violations of this Agreement to the fullest legal extent. npm may notify and cooperate with law enforcement authorities in prosecuting violations of this Agreement.

## Your Account

You must create and log into an Account to access features of some npm Services, including npm Open Source.

To create an Account, you must provide certain information about yourself, as required by the account creation form on the Website or the CLI. If you create an Account, you will provide, at a minimum, a valid e-mail address. You will keep that e-mail address up-to-date. You will not impersonate any other individual. You may delete your Account at any time by sending an e-mail to [support@npmjs.com](mailto:support@npmjs.com).

You will be responsible for all action taken using your account, whether authorized by you or not, until you either close your account or give npm notice that the security of your Account has been compromised. You will notify npm immediately if you suspect the security of your Account has been compromised. You will select a secure password for your Account. You will keep your password secret.

npm may restrict, suspend, or terminate your Account according to the Copyright Policy, if npm reasonably believes that you are in breach of these Terms, or if npm reasonably believes that you have misused npm Services.

## Your Content

Nothing in this Agreement gives npm any ownership rights in intellectual property that you share with npm Services, such as your Account information or any Packages you share with npm Services (*Your Content*). Nothing in this Agreement gives you any ownership rights in npm intellectual property provided via npm Services, like software, documentation, trademarks, service marks, logotypes, or other distinguishing graphics.

Between you and npm, you remain solely responsible for Your Content. You will not wrongly imply that Your Content is sponsored or approved by npm. npm will not be obligated to store, maintain, or provide copies of your content, except per the Privacy Policy.

npm may remove Your Content from npm Services without notice if npm suspects Your Content was submitted or used in violation of "Acceptable Use", as well as per the Copyright Policy.

You own Your Content, but grant npm a free-of-charge license to provide Your Content to users of npm Services. That license allows npm to make copies of and publish Your Content, as well as to analyze Your Content and share results with users of npm Services. npm may run computer code in Your Content to analyze it, but the license does not give npm any additional rights to run your code for its functionality in npm products or services. The license lasts, for each piece of Your Content, until the last copy disappears from npm's backups, caches, and other systems, after you delete it from the Website or the Public Registry.

Others who receive Your Content via npm Services may violate the terms on which you license Your Content. You agree that npm will not be liable to you for those violations or their consequences.

## Feedback

npm welcomes your feedback and suggestions for npm Services. You agree that npm will be free to act on feedback and suggestions you provide without further notice, consent, or payment. You will not submit feedback or suggestions that you consider confidential or proprietary.

## Indemnity

You will indemnify npm, its officers, directors, employees, representatives, and agents, and hold them harmless for, all liability, expenses, damages, and costs from any third-party claims, demands, lawsuits, or other proceedings alleging that Your Content, your use of npm Services, or both, violate the intellectual property right of a third party, this Agreement, or applicable law. You will not settle any such proceeding without the prior written consent of npm. npm will notify you of any such proceeding it becomes aware of.

## Disclaimers

*Use of npm Services is at your sole risk. npm Services are provided on an "as is" and "as available" basis. npm expressly disclaims all warranties of any kind, whether express, implied, or statutory, including implied warranties of title, noninfringement, merchantability, and fitness for a particular purpose.*

*npm makes no warranty that npm Services will meet your requirements, operate in an uninterrupted, timely, secure, or error-free manner, or that errors in npm Services will be corrected.*

*You receive material via npm Services at your sole risk. You will be solely responsible for any damage to your computer system and network, as well as any data loss that may result from use of npm Services or material received via npm Services.*

npm Services may provide information and software that is inaccurate, incomplete, misleading, illegal, offensive, or otherwise harmful. npm may, but does not promise to, review content provided by npm Services.

npm Services provide information about ownership and licensing of Packages, as provided by those Packages' publishers. That information may be wrong. npm cannot and does not provide legal advice.

### Third-Party Services

npm Services may hyperlink to and integrate with third-party applications, websites, and other services. You decide whether and how to use and interact with such services. npm does not make any warranty regarding such services or content they may provide, and will not be liable to you for any damages related to such services. Use of such third-party services may be governed by other terms and privacy notices that are not part of this Agreement and are not controlled by npm.

## Limits on Liability

*Neither npm nor any third-party service provider used by npm to provide npm Services will, under any circumstances, be liable to you for any indirect, incidental, consequential, special, or exemplary damages related to your use of npm Services or this Agreement, whether based on breach of contract, breach of warranty, tort (including negligence, product liability, or otherwise), or any other pecuniary loss, and whether or not npm has been advised of the possibility of such damages.*

*To the maximum extent permitted by law, npm's liability to you for any damages related to this Agreement, for any one or more causes and regardless of the form of action, will not exceed \$50.*

Some jurisdictions do not allow exclusion of certain warranties or limits on liability for incidental or consequential damages. Some of "Disclaimers" and "Limits on Liability" may not apply to you.

## Termination

Either you or npm may terminate this Agreement at any time with notice to the other.

On termination of this Agreement, your permission to use npm Open Source, as well any permission you may have to access Paid Services under additional terms, also terminate.

The following provisions survive termination of this Agreement: "Your Content", "Feedback", "Indemnity", "Disclaimers", "Limits on Liability", and "General Terms". Users of npm Services may continue to copy and share Your Content after termination of this Agreement.

## Payment Terms

There is no charge for use of npm Open Source. If you use Paid Services, these payment terms apply.

When enabling Paid Services, you must provide all the payment card details requested by the Website (your *Payment Details*). Those details must be for a valid payment card that you have the right to use (your *Payment Card*). You must keep your Payment Details up-to-date via the Website.

You can disable Paid Services at any time via the Website. npm will not refund any payment you have already made for Paid Services when you disable Paid Services.

Dollar amounts throughout this Agreement are amounts of United States Dollars. You must pay for Paid Services in United States Dollars.

Dollar amounts throughout this Agreement do not include tax. You will pay any tax.

## General Terms

If a provision of this Agreement is unenforceable as written, but could be changed to make it enforceable, that provision should be modified to the minimum extent necessary to make it enforceable. Otherwise, that provision should be removed.

You may not assign this Agreement. npm may assign this Agreement to any affiliate of npm, any third party that obtains control of npm, or any third party that purchases assets of npm relating to npm Services. Any purported assignment of rights in breach of this provision is void.

Neither the exercise of any right under this Agreement, nor waiver of any breach of this Agreement, waives any other breach of this Agreement.

This Agreement, together with the additional terms for Paid Services and npm software that you and npm agree to, embody all the terms of agreement between you and npm about npm Services. This Agreement supersedes any other agreements about npm Services, written or not.

## Disputes

The law of the State of California will govern any dispute, including any legal proceedings, relating to this Agreement or your use of npm Services (a *Dispute*).

You and npm will seek injunctions related to this agreement only in state or federal court in San Francisco, California. Neither you nor npm will object to jurisdiction, forum, or venue in those courts.

*Other than to seek an injunction, you and npm will resolve any Dispute by binding American Arbitration Association arbitration. Arbitration will follow the AAA's Commercial Arbitration Rules and Supplementary Procedures for Consumer Related Disputes. Arbitration will happen in San Francisco, California. You will settle any Dispute as an individual, and not as part of a class action or other representative proceeding, whether as the plaintiff or a class member. No arbitrator will consolidate any Dispute with any another arbitration without npm's permission.*

Any arbitration award will include costs of the arbitration, reasonable attorneys' fees, and reasonable costs for witnesses. You or npm can enter arbitration awards in any court with jurisdiction.

## Notices and Questions

You may send notice to npm and questions about the terms governing npm products and services by mail to npm, Inc., Legal Department, 1999 Harrison Street, Suite 1150, Oakland, California 94612, or by e-mail to [legal@npmjs.com](mailto:legal@npmjs.com). npm may send you notice using the e-mail address you provide for your Account or by posting a message to the homepage or your Account page on the Website.

---

Last modified February 02, 2016

Found a typo? Send a [pull request!](#)

## npm Organization Payment Plan

This npm Organization Payment Plan (this *Payment Plan*) supplements the terms for npm Open Source offered by npm, Inc. (*npm*) at <https://www.npmjs.com/policies/open-source-terms> (*npm Open Source Terms*), as well as the terms for npm Private Packages (*npm Private Packages*) at <https://www.npmjs.com/policies/private-terms> (*npm Private Packages Terms*). This Payment Plan governs payment for npm Private Packages organizations (*Organizations*) and use of npm Private Packages by user accounts added as members of those Organizations.

This Payment Plan was last updated on December 1, 2015. You can review prior versions at <https://github.com/npm/policies/commits/master/personal-plan.md>.

Under this Payment Plan, you may create one or more Organizations.

You will pay a minimum of \$14.00 via your Payment Card when you create an Organization and at the end of each month-long period, until you delete the organization. This minimum payment entitles you to add up to two user accounts not otherwise entitled to use npm Private Packages as member of the organization (*New Private Packages Accounts*), and an unlimited number of other user accounts. You will pay \$7.00 via your Payment Card per each additional New Private Package User (above two) that you add to an Organization, when you add them as a member and at the end of each month-long period.

The account holders of New Private Packages Accounts that you add as members of your Organizations will be entitled to use npm Private Packages under this Payment Plan, provided the account holders agree to and abide by the npm Open Source Terms and npm Private Packages Terms. These rights will not limit npm's right to enforce the npm Open Source Terms, npm Private Packages Terms, or other terms.

---

Last modified February 02, 2016

Found a typo? Send a [pull request!](#)

## npm Personal Payment Plan

This npm Personal Payment Plan (this *Payment Plan*) supplements the terms for npm Open Source offered by npm, Inc. (*npm*) at <https://www.npmjs.com/policies/open-source-terms> (*npm Open Source Terms*), as well as the terms for npm Private Packages (*npm Private Packages*) at <https://www.npmjs.com/policies/private-terms>. This Payment Plan governs payment for use of npm Private Packages by a single user account.

This Payment Plan was last updated on December 1, 2015. You can review prior versions at <https://github.com/npm/policies/commits/master/personal-plan.md>.

You will pay \$7.00 via your Payment Card when you enable npm Private Packages for your Account by selecting this Payment Plan, and at the beginning of each successive month-long period while this Payment Plan remains selected for your Account.

---

Last modified February 02, 2016

Found a typo? Send a [pull request!](#)

## npm Privacy Policy

We track data about users of the npm website, the npm public registry and any other services we may offer from time to time.

This page tells you what we track, and what we do with it.

## The npm Registry

All requests to the registry are logged and retained by npm, Inc. These requests include non-personally identifiable information such as the package requested and the time of the request, as well as potentially personally identifiable information such as the IP address of the requestor.

For example, this is a sample of the kind of data we track in our logs when you download a package:

```
2014-05-05T23:17:52Z 50.1.57.179 "-" "GET /npm-test-blerg" 200 "npm/1.4.10 node/v0.10.26 darwin x64" "install npm-test-l
2014-05-05T23:17:53Z 50.1.57.179 "-" "GET /npm/public/registry/npm-test-blerg/_attachments/npm-test-blerg-0.1000.1.tgz"
```

This includes a number of things:

1. The date and time.
2. The IP address of the requestor.
3. The HTTP request URL and method.
4. The HTTP response status code.
5. The **user-agent** string, which includes the versions of Node and npm in use.
6. Data about the CDN cache.
7. A random **npm-session** header, unique to a single invocation of the **npm** command line utility.
8. A **referrer** header, which will indicate the command that was invoked.

For example, if you type **npm install express**, then all HTTP requests as a result of that command will indicate that they are related to a single action, and that the originating request was for the **express** module.

Note that different versions of npm may send different information, so some of the fields may not be tracked for all requests.

## The npm Website

Like most website operators, npm, Inc. collects non-personally-identifying information of the sort that web browsers and servers typically make available, such as the browser type, language preference, referring site, and the date and time of each visitor request. npm, Inc.'s purpose in collecting non-personally identifying information is to better understand how npm's visitors use its website. From time to time, npm, Inc. may release non-personally-identifying information in the aggregate, e.g., by publishing a report on trends in website usage.

npm, Inc. also collects potentially personally-identifying information like Internet Protocol (IP) addresses. npm, Inc. does not use such information to identify its visitors, however, and does not disclose such information, other than under the same circumstances that it uses and discloses personally-identifying information, as described below.

The npm website uses Google Analytics to monitor and analyze user behavior. This service provides npm, Inc. with information on users' demographics, age, location, and interest categories, when such information is available. This information is not used to identify individual users, but can in some cases be very specific. You can learn more about the information gathered and retained by this service at the [Google Analytics privacy policy](#). You can opt out of Google Analytics entirely with the [Google Analytics opt-out browser add-on](#).

The npm website uses Oracle Marketing Cloud to collect and manage leads for marketing purposes. Information on what data Oracle Marketing Cloud stores about users, how it is collected, and how it is used are available in [Oracle Marketing Cloud & Oracle Data Cloud Privacy Policy](#).

The npm website uses Optimizely to test the effects of changes to the npm website on users' browsing experiences. Information on what data Optimizely stores about users, how it is collected, and how it is used are available in [Optimizely' privacy policy](#).

## Collection of Personally Identifying Information

In order to write information into the npm registry database (for example, to publish packages, bookmark packages, edit metadata, etc.) users may decide to provide certain personally identifying information including but not limited to: email address, username, password, personal website, and account names on other services such as GitHub, Twitter, and IRC.

When packages are published in the npm registry, the user responsible for the publish action is saved, along with the date and time of the publish. This information is shared on the website.

If you create an account or publish a package, your email address will be publicly disclosed.

If users do not want their information tracked in this manner, they can opt to not create an account. However, this means that some features of npm and the npm website will be unavailable to them.



# Use of Personally Identifying Information

We may use personally identifying information we have collected about you, including your email address, to provide you with news, notes, and recommendations. You can opt out of receiving such messages at any time by using the "unsubscribe" links or directions at the ends of messages you receive. In addition, we use collected personally identifying information to operate our business and the npm service. We do not disclose your personal information to unaffiliated third parties who may want to offer you their own products and services unless you have requested or authorized us to do so.

We may share your personal information with third parties or affiliates where it is necessary for us to complete a transaction or do something you have asked us to do. Likewise, we may share your personal information with third parties or affiliates with whom we have contracted to perform services on our behalf. Companies that act on our behalf are required to keep the personal information we provide to them confidential and to use the personal information we share only to provide the services we ask them to perform.

In addition, we may disclose personal information in the good faith belief that we are lawfully authorized to do so, or that doing so is reasonably necessary to comply with legal process or authorities, respond to any claims, or to protect the rights, property, or personal safety of npm, our users, our employees, or the public. In addition, information about our users, including personal information, may be disclosed or transferred as part of, or during negotiations of, any merger, sale of company assets, or acquisition.

## Cookies

A cookie is a string of information that a website stores on a visitor's computer, and that the visitor's browser provides to the website each time the visitor returns.

The npm website uses cookies to help identify and track visitors, their usage of the npm website, and their website access credentials. npm website visitors who do not wish to have cookies placed on their computers should set their browsers to refuse cookies before using npm, Inc.'s websites, with the drawback that certain features of npm, Inc.'s websites may not function properly without the aid of cookies.

## Packages Published to The npm Registry

Most packages published to the npm registry are open source, and freely available to all users of the npm service. We show basic package metadata on the npm website, in a variety of forms, so as to assist users in finding a package that meets their needs.

We may also inspect the contents of published packages to investigate any claims of malicious contents, or to debug problems that may occur in the process of running the service. For open source packages, we may also analyze the contents of published packages in an automated fashion to gain information about how people use npm packages. This information may be disclosed to third parties on our website, or in other forms. (Note that it is already freely available to anyone who downloads the packages themselves.)

If a package is published to the npm registry in such a way as to restrict read-access to the package, then we may still need to inspect the package contents on rare occasions. However, we never disclose information about a private package—including the fact that the package exists—to third parties who are not granted access to the package by the package's owners.

## Disclosure of Log Information

All user information is retained in raw form for such time as deemed appropriate by npm, Inc. It is shared with employees and contractors of npm, Inc., as needed to process information on npm, Inc.'s behalf.

Raw log data is not shared with third parties, but may be shared in aggregate. For example, every page on the npm includes a report on the number of downloads that module has received, and occasionally npm, Inc. may publish blog posts or reports on registry or website usage.

We also analyze log data for a variety of reasons, including counting up downloads and unique visitors, debugging production problems, tracking which versions of Node.js and npm are in use in the wild, and researching how npm packages are used together with one another. This helps us to better understand the usage patterns of npm, and make better decisions about the npm product.

## Use by Minors

The npm service is not intended for use by minor children (under the age of 18). Parents and guardians should monitor the use of the npm service by minor children. Children under age 13 should not use the npm service at all. If a child under age 13 submits personal information through any part of the service, and we become aware that the person submitting the information is under age 13, we will attempt to delete the information as soon as reasonably possible.

## Links to Other Websites

The npm service may contain links to other websites. Any personal information you provide on the linked pages is provided directly to that third party and is subject to that third party's privacy policy. Except as described above, we are not responsible for the content or privacy and security practices and policies of websites to which we link. Links from the npm service to third parties or to other sites are provided for your convenience. We encourage you to learn about their privacy and security practices and policies before providing them with personal information.

## United States Jurisdiction

The npm service is hosted in the United States. This Privacy Policy is intended to comply with privacy laws in the United States and may not comply with all privacy laws in other countries.

If you are a non-US user of the service, by using our service and providing us with data, you acknowledge, agree and provide your consent that your personal information may be processed in the United States for the purposes identified in this Privacy Policy. In addition, such data may be stored on servers located outside your resident jurisdiction, which may have less stringent privacy practices than your own. By using the npm service and providing us with your data, you consent to the transfer of such data and any less stringent privacy practices.

## Contact Information

If you have any questions or concerns about how we track user information, or how that information is used, please contact us at once.

You may contact npm, Inc. via our [contact form](#), by emailing [legal@npmjs.com](mailto:legal@npmjs.com), or via snail mail at:

npm, Inc.  
1999 Harrison Street  
Suite 1150  
Oakland CA 94612  
USA

## Privacy Policy Changes

Although most changes are likely to be minor, npm, Inc. may change its Privacy Policy from time to time, and in npm, Inc.'s sole discretion. Any such changes will be posted on [the npm blog](#), and the detailed history of changes can be found in [the git repository history for this document](#).

npm, Inc. encourages visitors to frequently check this page for any changes to its Privacy Policy. Your continued use of the npm website and the npm public registry after any change in this Privacy Policy will constitute your acceptance of such change.

## Credit and License

Parts of this policy document were originally included in the [WordPress.org privacy policy](#), used with permission.

This document may be reused under a [Creative Commons Attribution-ShareAlike License](#).

---

Last modified February 02, 2016      Found a typo? Send a [pull request!](#)

## npm Private Packages Terms

These npm Private Packages Terms of Use (these *npm Private Packages Terms*) supplement the terms for npm Open Source offered by npm, Inc. (*npm*) at <https://www.npmjs.com/policies/open-source-terms> (*npm Open Source Terms*). They govern access to and use of *npm Private Packages*, the private package storage, delivery, organization management, and access control features of <https://www.npmjs.com> (the *Website*) and the npm public registry at <https://registry.npmjs.org> (the *Public Registry*).

These npm Private Packages Terms were last updated on December 1, 2015. You can review prior versions at <https://github.com/npm/policies/commits/master/private-terms.md>.

You may only access or use npm Private Packages by agreeing to the npm Open Source Terms as supplemented by these npm Private Packages Terms. If npm adds any additional functionality to npm Private Packages, you must agree to these npm Private Packages Terms to use those new features, too. You add these npm Private Packages Terms to your agreement with npm by using npm Private Packages with your account (your *Account*). These npm Private Packages Terms then become a part of the contract between you and npm, until you or npm disable npm Private Packages for your Account.

## Use of npm Private Packages

npm will provide the private package storage and delivery features and services described in the public documentation for npm Private Packages at <https://www.npmjs.com/private-modules> (the *npm Private Packages Documentation*). npm grants you permission to use those features and services.

npm will also provide the organization management and access control features described in the npm Private Packages Documentation, and grants you permission to use those features and services, for npm Private Packages "organizations" to which your Account belongs.

Permission to use npm Private Packages is not exclusive to you, and you may not transfer it to others. These npm Private Packages Terms do not give you permission to give others rights to use npm Private Packages. If you agree to a Payment Plan that gives you that right, you may do so only according to that Payment Plan.

## Payment for npm Private Packages

Both your permission to use npm Private Packages and npm's commitment to provide npm Private Packages are subject to these npm Private Packages Terms, the npm Open Source Terms, and payment for use of npm Private Packages by your Account under a *Payment Plan*. Payment plans include:

1. the npm Personal Payment Plan at <https://www.npmjs.com/policies/personal-plan>
2. or the npm Organization Payment Plan at <https://www.npmjs.com/policies/organization-plan>

You may not use npm Private Packages unless you or someone else has agreed to a Payment Plan, enabled npm Private Packages for your Account under that Payment Plan, and made payment.

---

Last modified February 02, 2016      Found a typo? Send a [pull request!](#)

## policies

These are the legal policies of npm, Inc.

- [Terms of Use](#)
- [Code of Conduct](#)
- [Package Name Disputes](#)
- [npm License](#)
- [Privacy Policy](#)
- [Receiving Abuse Reports](#)
- [Copyright and DMCA Policy](#)
- [Trademark Policy](#)
- [Security](#)
- [Recruiting Process](#)

These are updated from time to time. Their sources are stored in a git repository at <https://github.com/npm/policies>.

---

Last modified February 02, 2016      Found a typo? Send a [pull request!](#)

## Receiving Reports

This is a guide for npm staff for handling user reports of harassment, package name disputes, and other user-generated pain points.

It is shared publicly in order to add transparency in our process

Nothing in this document should be considered a hard and fast rule in cases where it runs contrary to npm's mission of creating a safe, inclusive, and productive platform for easily sharing JavaScript modules.

When in doubt, seek help from your fellow admin staff.

## Package Name Disputes

If someone would like to take over a module name from another user, and asks for help with this, please refer them to the "Dispute Resolution" documentation. In cases like this, the two parties tend to be relatively rational and professional, and it is best if we encourage things to continue in that direction.

The policy in brief, where "Alice" is the original author, and "Yusuf" is the person with the dispute:

1. Yusuf emails Alice, explaining the reasons for wanting the module name, and CC's npm support.
2. We set a timer for 4 weeks. If that lands on a holiday or something, err on the side of making the delay longer.
3. At this point, one of three things have happened:
  - a. Alice and Yusuf have resolved the situation in a way that works for both of them.
  - b. Alice and Yusuf have reached an impasse, and cannot resolve the dispute.
  - c. Alice has not responded to Yusuf at all.

By far, (a) is the most common occurrence, and the answer is simple: we do nothing. This is ideal.

**When package name disputes can be handled amicably between users without any administrative involvement, everyone feels better about it.** Everything we do in these cases should guide towards that endgame when possible.

If Alice has not responded, then we must make a judgment call. There are a few possible considerations:

1. It could be that Alice has moved on to some other platform, doesn't care, and doesn't check this email address anymore, passed away, joined a monastery, meant to respond and forgot, who knows.
2. Alice has decided that she's never going to hand the module name over, so there's no point even discussing it.

Abrupt dismissive autocratic administration has a way of upsetting people and bringing them out of hiding. We cannot safely assume that absence is evidence of apathy.

1. Check the source control repository to see if Alice is writing patches, closing issues, etc.
2. Check to see if Alice has published new versions of the module.
3. Check to see if the module still works with the current version of Node, has a lot of dependents, etc.

If the module does appear abandoned, or if Yusuf's claim on the module is valid and that the hand-off would be less of a disruption than leaving an abandoned module in npm, then do this:

1. Reply-all to the thread with something like the following message, customized for your own "voice" as necessary:

```
Alice,

I hope that you are well, and that you only missed this
message because your life is too full of wonderful
distractions to be bothered dealing with this issue.

Yusuf is eager to take over the `fooblx` module on npm, and
plans to actively develop it.

Currently, it is [not actively developed, marked as
deprecated, apparently abandoned, not compatible with the
latest Node versions, whatever], so I'd like to hand it off to
Yusuf to take over.

If this causes you any stress or inconvenience, please let me
know as soon as possible.
```

2. Set a timer for 1 week.
3. If Alice responds with concerns, then use diplomacy. Usually this comes down to telling Yusuf, "Sorry, you'll have to choose another name." Mostly, people are pretty receptive to this.
4. If Alice still does not reply, then reply-all to the thread with something like this:

```
Alice,  
  
As per the email last week regarding the fooblx module, we've  
decided to hand control over to Yusuf, who will be actively  
developing it.  
  
Yusuf,  
  
You are now a package maintainer on the fooblx module.  
  
Please leave the existing versions in place, and bump the  
major version, so that any prior users are minimally impacted.  
  
Thank you both for your patience and understanding.  
  
--i
```

Caveats and things to be sensitive of:

- People are often surprisingly attached to the names that they give their code.
- If someone feels like they have an opportunity to be heard, they are much more likely to feel like the process is fair, even if they don't ultimately get what they want.
- Compassion and respect are pragmatic time-saving tools that prevent unnecessary pain and hardship for our users. Use these tools, always.

Note that this does not mean that we will always try to accommodate users' wishes. If a module name is offensive, the package contents are violating licenses or other intellectual property rules that could get us in trouble, or the package is empty (i.e., squatting), or otherwise violates the terms of use, we reserve the right to remove packages without any discussion.

Even in those cases, it is often best to try to give users a week or so to do things on their own, so that they can maintain a sense of ownership. Outright and obvious trolling or harassment is never tolerated, however.

## Harassment Reports

These are cases where a user is reporting to us that someone is using the npm system for nefarious ends, or harassing other users in some other way.

In this case, we draw a very hard line, as communicated by our zero-tolerance anti-harassment policy.

Reports of abuse *of npm* are somewhat different than reports of abuse *of npm users*.

### Reports of Violations of the npm Terms of Service

If a user is publishing a flood of empty squatting packages, spamming, phishing, offensive content, or other childish trolling aimed at the service rather than at a specific user, then the course of action is simple:

1. Ban the user.
2. Clean up the mess.

If it's possible that they are unaware that their behavior is not allowed, it is a good idea to not ban the user outright, but send them an email asking them to please stop the bad behavior.

Here's an example:

```
Subject: Empty/duplicate packages removed  
From: Isaac Schlueter <isaacs@npmjs.com>  
To: Some User <some-user@gmail.com>  
Cc: npm support <support@npmjs.com>
```

Several empty and duplicated packages belonging to you were removed.

Please do not publish empty packages to npm. This causes difficulty for others who may want to use names for new projects.

We do not allow "reserving" names for future use. You must have something to publish before taking a package name. Otherwise, we quickly end up with a lot of empty packages, and names being used for no purpose.

If you continue to publish empty packages to npm, your username and/or IP address may be blocked from accessing the service.

Thank you.

-- i

Do not mention, involve, or CC the person who reported the bad behavior, as this can only result in added conflict. Briefly thank them for the report, and let them know that it's been dealt with.

## Reports of Targeted Harassment

This includes both abuse of npm users via the npm service, as well as auxiliary channels such as IRC, Twitter, GitHub, etc.

**If it impacts npm users and degrades their experience of using the service, then it's our problem, and we take it seriously.**

The vast majority of reports of harassment will come via written media (email, IRC, etc.) If you receive a report of harassment in a non-text format, ask the user for a written account if this is reasonable. If it is not, then take your own notes, or record it in a written format as soon as possible.

A verbal report lasting more than a minute or so is probably better conducted in a quiet/private place rather than in a public space, for the safety and comfort of the reporter. This also decreases the chances for someone to overhear sensitive information that the reporter may not want spread around at an event.

If the user would prefer to remain anonymous, please strip their name from the record prior to sharing it with the rest of the abuse team.

Try to get as much detail as you can about the incident. This will assist us later if we ever need to make a case to defend our choices, as well as inform future decisions about how these incidents could be avoided.

If the following information is not volunteered in the written or verbal report, ask for it/include it, but do not pressure them.

- Identifying information (user name, email address, etc.) of the user doing the harassing.
- The behavior that was in violation.
- The approximate time of the behavior (if different than the time the report was made).
- The circumstances surrounding the incident.
- Other people involved in the incident.

Generally we are not equipped for evidence gathering: do not go around "interviewing" others involved.

### Threats to physical well-being

If someone reports that a user of the service or an attendee at an event has committed or is threatening violence towards another person, or other safety issues:

- If there is any general threat to our users or the safety of anyone including npm staff is in doubt, summon security or police.
- Offer the victim a private place to sit.
- Ask "is there a friend or trusted person who you would like to be with you?" If so, arrange for someone to fetch this person.
- Ask them "how can I help?"
- Provide them with your list of emergency contacts if they need help later.
- Do not touch the victim to console them unless they initiate. It can make things worse.

### Law enforcement

If everyone is presently physically safe, involve law enforcement or security only at a victim's request.

In many cases, reporting harassment to law enforcement is very unpleasant and may result in further harassment. Forcing victims to go to law enforcement will reduce reports of harassment (but not actual harassment). For more information, see [Why Didn't You Report It?](#)

A staff member can provide the list of emergency contacts and say something like "if you want any help reporting this incident, please let us know" and leave it at that.

### Reports of harassment that were widely witnessed

These include things like harassing content in package names, conference talks, or harassment that took place in a crowded space.

Simply say "Thanks, this sounds like a breach of our anti-harassment policy. I am going to convene a meeting of a small group of people and figure out what our response will be."

Often, the best approach is similar to handling package name disputes. For example, a user may be a non-native English speaker, and not realize that a given term is offensive. It is our responsibility as the caretakers of npm to attempt to resolve this as amicably as possible.

Other times, this may be a matter of simply deleting some offensive packages and telling the user not to do it again.

In the most egregious cases, it may require banning the user account and/or IP address of an abusive troll.

### Reports of more private harassment

Offer the reporter/victim a chance to decide if any further action is taken: "OK, this sounds like a breach of our anti-harassment policy. If you're OK with it I am going to convene a meeting of a small group of people and figure out what our response will be."

Pause, and see if they say they do not want this. Otherwise, go ahead.

**Do not:**

- Overtly invite them to withdraw the complaint or mention that withdrawal is OK. This suggests that you want them to do so, and is therefore coercive. "If you're OK with it [pursuing the complaint]" suggests that you are by default pursuing it and is not coercive.
- Ask for their advice on how to deal with the complaint: this is our responsibility.
- Offer them input into penalties: this is our responsibility.
- Share details of the people involved or incident without specific permission from the victim. This includes sharing with other staff.

**Staff action in response to harassment reports**

We should aim to take action as soon as reasonably possible. During the event, a response within the next half-day is usually an appropriate timeframe. After the event you may need more time to gather sufficient decision makers, but ideally responding within the same week or sooner is good.

**Meeting**

Available staff should meet as soon as possible after a report to discuss:

- What happened?
- Are we doing anything about it?
- Who is doing those things?
- When are they doing them?

Neither the complainant nor the alleged harasser should attend. (If the event was very widely witnessed, such as a harassing talk, this may be an exception to this guideline.) People with a conflict of interest should exclude themselves or if necessary be excluded by others.

**Communicate with the alleged harasser about the complaint**

As soon as possible, either before or during the above meeting, let the alleged harasser know that there is a complaint about them, let them tell someone their side of the story and that person takes it into the meeting.

**Communicate with the harasser about the response**

As soon as possible after that meeting, let the harasser know what action is being taken. Give them a place to appeal to if there is one, but in the meantime the action stands. "If you'd like to discuss this further, please contact XYZ, but in the meantime, you must <something something>"

**Don't require or encourage apologies**

Do not ask for an apology to the victim. We have no responsibility to enforce friendship, reconciliation, or anything beyond lack of harassment between any two given users, and in fact doing so can contribute to someone's lack of safety while using our service.

Forcing a victim of harassment to acknowledge an apology from their harasser forces further contact with their harasser. It also creates a social expectation that they will accept the apology, forgive their harasser, and return their social connection to its previous status. A person who has been harassed will often prefer to ignore or avoid their harasser entirely. Bringing them together with a third party mediator and other attempts to "repair" the situation which require further interaction between them should likewise be avoided.

If the harasser offers to apologize to the victim (especially in person), strongly discourage it. In fact, discourage *any* further interaction with the offended party.

If a staff member relays an apology to the victim, it should be brief and not require a response. ("X apologizes and agrees to have no further contact with you" is brief. "X is very sorry that their attempts to woo you were not received in the manner that was intended and will try to do better next time, they're really really sorry and hope that you can find it in your heart to forgive them" is emphatically not.)

If the harasser attempts to press an apology on someone who would clearly prefer to avoid them, or attempts to recruit others to relay messages on their behalf, this may constitute continued harassment.

**Data retention**

All (potentially de-identified) information about harassment reports should be stored for a period of at least 5 years, in an electronic format, accessible only by the npm abuse team.

Lifetime bans are handled by banning a username or IP address. If it ever becomes necessary, we will maintain a lifetime ban of users for in-person events as well.

## Communicating with the npm community

In general, we handle disputes and harassment quietly. Our code of conduct explicitly forbids harassment, and we maintain our reputability on this point by enforcing that policy appropriately.

However, occasionally these events will spill out into public. In those cases, please let the npm executive team decide how best to communicate with the public.

### Principles of public communication

- Show that the npm anti-harassment policy is being enforced fairly.
- Explain (briefly, neutrally, anonymously) what violation led to the enforcement action. For example, "A package with an offensive name was removed by the npm staff, after reporting the issue to the author."
- Help the community understand that they are not in danger of being victimized by capricious or irrational administrative actions.

When necessary, this will be communicated via the npm blog.

When it's necessary to communicate enforcement of our policy at an in-person event, a brief public statement to the attendees such as this would suffice:

"[thing] happened. This was a violation of our policy. We apologize for this. We have taken [action]. This is a good time for all attendees to review our policy at [location]. If anyone would like to discuss this further they can [contact us somehow]."

And then move on with the program.

### Dealing with upset users

People may be upset and wish to express their concerns to npm staff. We should be in "making the person feel heard" mode; it's important not to cross into "education mode". Hear them out, take notes as appropriate, thank them for their thoughts.

We should not share additional details of the incident with uninvolved parties.

If a user is upset and a staff member agrees that a wrong was done to them, it helps a lot to just say simply "I'm so sorry." (Rather than "but we tried really hard" or "no one told us" or etc., even if that was true. "I'm so sorry" goes a long way to defusing many people's anger.)

Whether or not a staffer agrees that a wrong was done to them, the user should be armed with an authority they can appeal to if talking wasn't enough. "Please email [abuse@npmjs.com](mailto:abuse@npmjs.com)."

## Evaluation

After we have had a chance to observe how the anti-harassment and dispute resolution policies work in the real situations, we may wish to change the policy to better address them.

Did anything unforeseen happen that there should be a rule about? Sometimes an unacceptable behavior does not warrant a whole new rule, but should be listed as a specific example of unacceptable behavior under an existing rule.

For the sake of consistency, if there are changes to a rule, we try to apply that rule moving forward, rather than retroactively. If a judgment call is made, record the decision and the justification, and perhaps codify it in a rule going forward so that users can more easily succeed in our community.

## Changes

This is a living document and may be updated from time to time. Please refer to the [git history for this document](#) to view the changes.

## Credit and License

Parts of this policy borrow heavily from the [Geek Feminism Wikia guide](#).

This document may be reused under a [Creative Commons Attribution-ShareAlike License](#).

## npm, Inc.'s Recruiting Process



This document was once an internal document at npm, Inc. Because it is relevant to many people in our community, and we like transparency, we have decided to open it up to the world.

There is no expectation or requirement that *you* hire people in this way if you are not working at npm, Inc. This is just how we do it. If you [apply for a job here](#), this may provide some clue as to what to expect.

From here on out, "you" refers to the npm, Inc. employee who is evaluating candidates for hire.

## npm's hiring philosophy

npm has a great brand for recruitment. Our applicants generally mention three things they particularly like:

- Our focus on work-life balance and a humane working environment
- Caring about diversity
- That our product is open source

None of these things have anything to do with node.js or JavaScript, and that's a great sign. Your conversations with applicants should communicate the value we place on these things.

## Writing the job description

Some simple guidelines:

- Keep a conversational tone. Imagine you were describing the job to somebody over IM. This is not a contract.
- Maintain a bias towards inclusion. Applying for jobs is scary and hard work, so people read job descriptions looking for the *first thing that disqualifies them* so they don't need to apply. Don't give them this opportunity. *The purpose of a job description is to persuade someone to apply, not to scare them off.*
- Don't list a qualification (e.g. a degree) as required unless it really is required (we do not require degrees).
- Likewise, don't list a qualification if it is "nice to have". If we get a lot of applicants, then we'll screen for stuff that is nice to have after we get their resumes.
- Do not include a laundry list of technologies or applications. Any smart hire will be able to learn these.
- Talk about how the role will grow, and opportunities for training and mentorship. The number one thing people look for in a job is the opportunity to be better at it.
- Avoid [aggressive language](#) of any kind.
- Gendered language is obviously a no-go, and consider if your language is ageist or otherwise biased.

We have an [open source repo](#) of former job descriptions to help you.

## Recruitment process

Role is identified.

Hiring Manager writes job description.

As resumes come into [jobs@npmjs.com](mailto:jobs@npmjs.com), HR will upload to and update [Lever](#). It is the responsibility of the hiring manager to review resumes on a timely basis and if there are obvious "NO's" to let HR know immediately so the candidates can move forward with their job search.

Everyone who speaks to a candidate must put their notes in [Lever](#). The #recruitment channel on slack is for discussing hiring pipeline status and recruitment plans, not individual candidates. (i.e. it's okay to ask "is Jane coming for an interview?" but not "I didn't like Jane because...").

1. Post the job description.
2. Wait at least 2 weeks after posting a job description before screening (don't screen as you go; screen as a batch).
3. All incoming applicants MUST be put into Lever. If you receive one in your personal email account, please put into Lever and let HR know. This ensures that every application gets a personal response indicating we received it and a rough timeline for screening (e.g. "about 1 week from now").
4. Hiring manager will read all available resumes and select 25-50% of those candidates for first-round screening. Anybody not selected is notified at this stage by HR.
5. First-round screening is an informal, 20-30 minute discussion, by phone, Skype, or Google Hangout by the hiring manager, who will select roughly 50% of these candidates for second-round screens. Anybody not selected is notified at this stage by HR.
6. Second-round screening is 2 longer conversations, 45-60 minutes, by phone or google hangout if the applicant is remote, or in-person if they are local. Hiring Manager will determine which prospective team-mates, or existing employees with relevant skillsets will be taking part in second-round conversations. For remote applicants always supply the phone number or google hangout name *applicant-interviewer* in the calendar invitation. Anybody not selected for this stage will be notified by HR.
7. After the second round interviews, hiring manager meets with interviewers to select 1-3 candidates for a full team interview. Often by this point there will be one obviously best candidate, and it is fine to bring them in by themselves. Candidates not selected for the final round are notified at this stage by HR.

8. Full team interview: the candidate comes in for lunch at noon (any day except Monday). The hiring manager will invite 6 staff members that the candidate is likely to be in contact with during their day. After lunch, anyone who hasn't yet spoken with the candidate, and the hiring manager deems necessary may request a 30-45 minute slot. For remote candidates after their flight, hotel etc. have been arranged HR will send them their itinerary and cc hiring manager, with a short "this might come in handy" note containing at least one contact number for emergencies and local cab or other transit options.
9. After the full team interview, hiring manager and their team will take everyone's notes into account and make a final decision. After selecting the best candidate, we do *not* yet notify the other candidates.
10. Reference checks: The candidate supplies 3 names and contact information of people they have worked directly with; hiring manager has a discussion with at least 2. This is a final red-flag check and also gives the hiring manager a head-start on how to work best with the hire. **Questions can include** What dates did the candidate work there? What is the documented departure reason? Would you rehire?
11. Hiring manager will consult with the selected candidate on salary, benefits, relocation, visa (at the moment npm lacks the legal resources to sponsor visas other than TN-1 visas for Canadian and Mexican citizens), and any other issues to make sure there are no unexpected barriers to a hire and a reasonable start date, and make a salary recommendation to the CEO.
12. CEO will have a final conversation with the candidate, make salary offer (we will always offer the best salary we can afford, so there is not much room for negotiation on salary) and discuss equity. If an agreement is reached, we prepare an offer letter. If not, we consider one of the other final-round candidates.
13. Once the candidate has *signed* and we have received the offer letter, HR will notify the other final round candidates.

## How we interview

Read [Laurie's blog post on hiring](#) for all the things *not* to do. The remainder of this section assumes you've read this post.

We are looking for people who:

- can grasp complex technical subjects, because our product is one of those
- can clearly explain those subjects to others, because simplifying the complex is what we do
- can get things done, the best indicator of this being having already done things
- can quickly learn how to do this job; they don't need to already know how to do it
- show signs that they will get better at this job, i.e. future growth potential

npm values transparency and humanity, so as much as possible our interviews are transparent and humane. Interviews are inherently uncomfortable and scary, and nobody sounds smart when they are uncomfortable and scared. Do your best to compensate for this. If you think the candidate is doing well, be liberal about saying so. If the candidate makes a mistake, try to prevent them spiraling into meltdown by moving on quickly or giving positive feedback about some other aspect of their performance.

Your interview should be a conversation. You are not trying to prove that you know more about a topic than they do. You are not trying to quiz them to make sure they know everything about a specific topic or technology. You want to know if they can grasp complex concepts and explain them clearly, because that is what a knowledge worker does.

## Changes

This is a living document and may be updated from time to time. Please refer to the [git history for this document](#) to view the changes.

## License

This document may be reused under a [Creative Commons Attribution-ShareAlike License](#).

---

Last modified February 02, 2016

Found a typo? Send a [pull request!](#)

## npm Security Policy

Outlined in this document are the practices and policies that npm applies to help ensure that we release stable/secure software, and react appropriately to security threats when they arise.

## Table of Contents

1. [Reporting Security Problems to npm](#)

2. [Security Point of Contact](#)
3. [Onboarding Developers](#)
4. [Separation of Duties and Authorization](#)
5. [Critical Updates And Security Notices](#)
6. [Responding to Security Threats](#)
7. [Vulnerability Scanning](#)
8. [Password Policies](#)
9. [Application Design Best Practices](#)
10. [Development Process](#)
11. [AntiVirus Software](#)

## Reporting Security Problems to npm

If you need to report a security vulnerability, please [contact us](#) or email [security@npmjs.com](mailto:security@npmjs.com).

We review all security reports within one business day. Note that the npm staff is generally offline for most US holidays, but please do not delay your report! Our off-hours support staff can fix many issues, and will alert our security point of contact if needed.

## Security Point of Contact

npm's CTO [Laurie Voss](#) is the current point of contact for all security related issues.

Any emails sent to [security@npmjs.com](mailto:security@npmjs.com) will be escalated to the security point of contact, who will delegate incident response activities as appropriate.

## Onboarding Developers

All new technical hires are introduced to our security policy as part of the onboarding process.

## Separation of Duties and Authorization

- Developers are only be given access to key npm services (Fastly, AWS, etc) when it's required for their job.
- [IAM](#) is used to limit the permissions on AWS accounts, minimizing the damage that would be incurred if an account is compromised.

## Critical Updates And Security Notices

We learn about critical software updates and security threats from a variety of sources:

- Ubuntu's security notices page: <http://www.ubuntu.com/usn/trusty/>
- The Node.js mailing list.
- The [security@npmjs.com](mailto:security@npmjs.com) email address.
- and other media sources.

### Ubuntu Automatic Security Updates

Along with keeping an eye out for critical security updates, automatic security updates are enabled on all of our production servers allowing patches to be applied immediately without human intervention.

<https://help.ubuntu.com/community/AutomaticSecurityUpdates>

## Responding to Security Threats and Critical Updates

When a security threat is identified, we have the following process in place:

1. We have the slack channel **security-all**, which is used to prioritize and coordinate responses to security threats.
2. Our [Security Point of Contact](#) oversees this discussion: managing the triage, responding to emails, and updating npm's status page.
3. Based on the triage, work is allocated to developers to address the threat:

- **P0** : Drop everything and fix!

- P1 : High severity, schedule work within 7 days.
- P2 : Medium severity, schedule work within 30 days.
- P3 : Low severity, fix within 180 days.

## Vulnerability Scanning

Along with reacting to security notifications as they happen, we proactively pen-test and audit software.

### Third-Party Audits

We perform regular penetration testing and code audits with the security firm [Lift Security](#).

While working on features at npm, all engineers coordinate security audits with the [Security Point of Contact](#).

Documents from this process are available, and can be provided to customers when requested.

### Automated Scanning

The cloud hosting platforms that we use provide options for automated vulnerability scanning.

- AWS: <https://aws.amazon.com/security/penetration-testing/>
- SoftLayer: <https://knowledgelayer.softlayer.com/procedure/nessus-security-scanner>

## Password Policies

- Enable 2FA on all npm related accounts.
- Passwords should be rolled every 90 days.
- Passwords should contain alpha-numeric characters and symbols.
- Passwords should be a minimum of 8 characters.
- Any systems we build that accept a username and password should reject a user after repeated failed login attempts.

### Don't Use Passwords

We should opt for alternative authentication methods when possible:

- Asymmetric keys for connecting to servers.
- Delegated authentication (SAML, OAuth2, etc).
- Opaque access tokens.

### SSH Keys

SSH keys should be rolled out selectively, providing developers access to only the servers that they require access to.

## Application Design Best Practices

In the next section of the document, we discuss the design methodologies that we use to build stable and secure software.

### Logging Practices

Logs are important for both debugging applications and detecting security breaches in our software -- ask CJ for a speech about logging.

#### What We Log

- We should track failed login attempts to servers:
  - Ubuntu provides this information in `/var/log/auth.log`
- We should log the operations performed by users:
- Ubuntu provides this information in `history`.
- Applications should provide detailed operational logs in a [standardized format](#).

#### Log format

All applications should contain logging for **date** , **time** , **operation** , and a **unique request identifier** .

We use [common-log-string](#) internally to standardize this:

### [Backing Up Logs](#)

At least 90 days of logs should be kept for each service. On high traffic hosts this may require backing-up logs in cloud storage on a regular basis.

### [Reviewing Logs](#)

On the servers that we manage for other companies, we should audit logs on a regular basis.

TODO: We plan to build automated anomaly detection systems in place for our logs [see internal issue #381](#).

### [Secrets in Logs](#)

Logs should not contain any sensitive user information, e.g., passwords.

The module [hide-secrets](#) is used to help with this.

## Limiting Access to Operating System Files

Micro-services should only have access to databases and files that they need access to.

With our docker-based infrastructure (npm On-Site) this is achieved by having containers only mount folders on the root host that they require access to.

In our production environment, this is achieved by partitioning services across multiple hosts.

## Security Groups

Security-groups, or Zones in the case of SoftLayer, are used to limit the network connectivity between hosts.

When deploying a service, ask: "what other services does this actually need to connect to?"

## Storage of Data

Any sensitive user information should be encrypted at rest. Using [encrypted EBS drives](#), or an equivalent, is a great way to achieve this.

## Inter-Service Communication

Communication between services on the same host can be performed via HTTP.

All inter-service communication between two hosts is performed using TLS.

# Development Process

npm has a well-defined, security-focused, development process:

## Code Reviews

No code goes into production unless it is reviewed by at least one other developer.

The onus is on the reviewer to ask hard questions: "what are the ramifications of opening up port-X?", "why is this connection being made over HTTP instead of HTTPS?"

## Deploying Updates

- Any new code pushed to production is first thoroughly tested in a staging environment.
- Mechanisms are in place for rolling back any changes that are pushed to production.
  - If a schema-change is involved, an inverse migration is first tested in staging (we want to be confident that we should roll things back).

## Unit Testing

We love testing at npm:

- During the code-review process, if you see logic that's complicated and lacks a test, politely ask the developer for a test.
- It's particularly important that tests are added to logic that interacts with sensitive parts of the system: ACL logic, password validation, database access.
- Tests should not contain user-data, make sure to anonymize email addresses, usernames, etc.

- Test coverage is a great way to make sure all of the nooks and crannies of your codebase are tested. npm maintains two tools for test coverage internally [tap](#), and [nyc](#).
- Any new functionality should always come with a test to verify that it does what we think it does.
- Any bug fix should always come with a test so that we don't have to encounter the same bug multiple times.

## Design Cycle

The design process, and management techniques vary from team to team at npm. Across the board, however, we strive to have continuous deployments. Releasing many small features as they become production ready.

Security is taken into account during all phases of the software development life-cycle: unit tests think about potential threats; when testing on staging, we attempt to test potential exploits, etc.

# AntiVirus Software

On our managed Ubuntu hosts, we run the [ClamAV](#) AntiVirus software.

## When A Virus Is Identified

The infected server should be retired, and a new server should be provisioned from scratch.

# Changes

This is a living document and may be updated from time to time. Please refer to the [git history for this document](#) to view the changes.

# License

This document may be reused under a [Creative Commons Attribution-ShareAlike License](#).

---

Last modified February 02, 2016

Found a typo? Send a [pull request!](#)

## Term and Licenses

npm, Inc. offers software and services under a few different licenses and terms of use.

# Software from npm

License terms and notices for the `npm` command-line program can be found in the LICENSE file of the project's source code at <https://www.github.com/npm/npm>.

# npmjs.com and Other npm Services

The npm Open Source Terms at <https://www.npmjs.com/policies/open-source-terms> govern use of <https://www.npmjs.com> and the npm public registry.

The npm Private Packages Terms at <https://www.npmjs.com/policies/private-terms> govern use of npm Private Packages.

The npm Personal Payment Plan <https://www.npmjs.com/policies/personal-plan> and the npm Organization Payment Plan <https://www.npmjs.com/policies/organization-plan> govern payment for npm Private Packages.

---

Last modified February 02, 2016

Found a typo? Send a [pull request!](#)

## npm Trademark Policy

### What is npm?

npm is a package manager for Node.js modules. It was created in 2009 by Isaac Schlueter as an open source project to help JavaScript developers share modules (aka packages) of code in a very easy and efficient way.

The npm project contains two main parts:

1. The npm client, which is bundled with the Node.js platform, and is used as a command line tool to install and publish packages. It is simply called "npm". Its code is available as an open-source project: <https://github.com/npm/npm>
2. The npm registry service. When a user of the npm client runs the command `npm install example-package`, the npm client retrieves the package and all related dependencies from the npm registry service. The example output looks like this:

```
user1-MacBook-Pro-2:~ user1$ npm install example-package
npm http GET https://registry.npmjs.org/example-package
npm http 200 https://registry.npmjs.org/example-package
example-package@2.4.1 node_modules/example-package
```

The registry is run as a free (as in beer) public service for anyone wanting to publish an open source package and for anyone to install an open source package.

### What is npm, Inc. and what is its mission statement?

npm, Inc. is a company co-founded by npm's creator, Isaac Schlueter, along with Laurie Voss and Rod Boothby.

npm, Inc. is dedicated to the long term success of the JavaScript community, which includes the success of the open-source Node.js and npm projects.

At npm, Inc. we do three things to support this goal:

1. Run the open source registry as a free service.
2. Build tools and operate services that support the secure use of packages in a private or enterprise context.
3. Build innovative new tools and services for the developer community.

### Why npm, Inc. has a trademark policy

npm, Inc. has filed for trademarks for the npm name and logo. We have developed this trademark usage policy with the following goals in mind:

- We'd like to make it easy for anyone to use the npm name or logo for community-oriented efforts that help spread and improve npm.
- We'd like to make it clear how npm-related businesses and projects can (and cannot) use the npm name and logo.
- We'd like to make it hard for anyone to use the npm name and logo to unfairly profit from, trick, or confuse people who are looking for official npm resources.

### Nominative use - No need to type <sup>TM</sup> on twitter

It's perfectly OK to use "npm" to refer to npm, Inc., to the npm software, and to the npm public registry. That's different from using npm in the name of one's own product or service.

Nominative use means it's OK to refer to something that is trademarked, but it is not OK to incorporate one company's trademark into another company's product, service, or company name. That's why you would need permission from the trademark owner to open "Discount Nike Shoes" or "iPad App Marketplace".

### The npm trademark policy

We ask that you get permission from npm, Inc. to use the npm name or logo as part of the name of any project, product, service, domain or company.

We will grant permission to use the npm name and logo for projects that meet the following criteria:

- The primary purpose of your project is to promote the spread and improvement of the npm client software or the npm registry service.
- Your project is non-commercial in nature (it can make money to cover its costs or contribute to non-profit entities, but it cannot be run as a for-profit project or business).

- Your project neither promotes nor is associated with entities that currently fail to comply with the Artistic License 2.0 under which npm is distributed, or which are in violation of this trademark policy.

For other projects, we have set up the following rules around the use of the npm trademark:

- A. It is not OK use npm in the name of any product, service, or project, except after the preposition "for". For example, "Private Registry Hosting for npm". You should not use npm in a company or domain name for a package registry or related service without our permission.
- B. When referring to the npm software in body text, the first usage should be followed by a generic term such as "package manager" to provide context. npm should never be used or explained as an acronym.
- C. When referring to the npm public registry, please follow npm with the word "registry" or the phrase "public registry".
- D. When referring to a private registry for npm packages, please describe it as "private registry for npm packages" or as a npm-registry proxy when first referring to it in the text.
- E. References to the owner of the npm client software and the operator of the npm public registry should be to npm, Inc.
- F. Any materials referring to npm should include the following notice in the fine print: "npm is a trademark of npm, Inc."

Examples of ways to use the term "npm" in a name

- <Your consulting co's> services for npm  
e.g. Pink Unicorn Consulting Ltd. services for npm
- <Your software co's> private registry for npm  
e.g. Purple Unicorn Inc. private registry server for npm

An example of a way to refer to npm when describing your solution:

- **Paypal/Kappa, a hierarchical npm-registry proxy**

In the case of Kappa, it is clear that the open source project name is "Kappa" so no one would be reasonably confused that it is the official npm client, or the official npm registry.

Explaining that something is a proxy of the npm registry is OK.

When in doubt about your use of the npm name or logo, please [contact npm, Inc.](#) for clarification.

## The npm Logo

Our npm Logo is very recognizable and deserves special treatment. The npm Logo signifies us, or a special relationship with us, and you should use it only with our permission. Since the goal is to avoid confusion about you being us, or your relationship with us, context counts. We will consider requests on a case-by-case basis.

## The npm Wombat Mascot

Like the npm Logo, the npm Wombat graphic is a very recognizable part of the npm brand, and signifies a special relationship with the the npm project, service, or company. It should never be used except with explicit written permission. We will consider requests on a case-by-case basis.

Please be advised that, unlike the npm logo, the Wombat generally may **not** be used to refer to the project, service, or company in a nominative sense, as any usage will almost always imply a special relationship with npm.

## Changes

This is a living document and may be updated from time to time. Please refer to the [git history for this document](#) to view the changes.

## License

Copyright (C) npm, Inc., All rights reserved

This document may be reused under a [Creative Commons Attribution-ShareAlike License](#).



## About npm

npm is lots of things.

- npm is the package manager for [Node.js](#). It was created in 2009 as an [open source project](#) to help JavaScript developers easily share packaged modules of code.
- The npm Registry is a public collection of packages of open-source code for Node.js, [front-end web apps](#), [mobile apps](#), [robots](#), [routers](#), and countless other needs of the JavaScript community.
- npm is the command line client that allows developers to install and publish those packages.
- npm, Inc. is the company that hosts and maintains all of the above.

## About npm, Inc.

npm, Inc. is a company co-founded in 2014 by npm's creator, Isaac Z. Schlueter, along with Laurie Voss and Rod Boothby.

npm, Inc. is dedicated to the long term success of the JavaScript community, which includes the success of the open-source Node.js and npm projects.

At npm, Inc. we do three things to support this goal:

1. Run the open source registry as a free service.
2. Build tools and operate services that support the secure use of packages in a private or enterprise context.
3. Build innovative new tools and services for the developer community.

npm's mission is to take Open Source development to entirely new places. When everyone else is adding force, we work to reduce friction.

npm is not a typical product, and we are not a typical "work hard/play hard" startup. We are responsible adults with diverse backgrounds and interests, who take our careers and our lives seriously. We believe that the best way to iterate towards success is by taking care of ourselves, our families, our users, and one another. We aim for a sustainable approach to work and life, because that is the best way to maximize long-term speed, while retaining clarity of vision. Compassion is our strategy.

Our offices are in downtown Oakland, California.

## Contacting npm

You can reach us for support or any other questions via our [contact form](#) or at [npm@npmjs.com](mailto:npm@npmjs.com).

---

Last modified January 11, 2016

Found a typo? Send a [pull request!](#)

## Nice People Matter: npm is hiring

If you would like to work with us and have skills you think would be helpful, please send an email with evidence of your experience (like your resume, or links to LinkedIn/GitHub/wherever) to [jobs@npmjs.com](mailto:jobs@npmjs.com).

We'd like to hear from you.

## Internships

If you're currently in full-time education (class of 2016 or 2017) and looking for a summer internship with an open-source project that can also pay you, we're happy to say that we can offer you a place at npm in partnership with our investors, as part of the [True Entrepreneur Corps](#) program.

We are looking primarily for enthusiasm and intellectual curiosity, but any experience with JavaScript and especially Node.js is obviously a big advantage.

You'll be working on the open-source npm project itself, so your work will be code-reviewed by the same people who run the project, your commits will immediately benefit tens of thousands of active users, and will also be publicly verifiable by future employers.

Last year's intern, [Faig](#), wrote about [what he learned](#) on TEC's blog and also had this to say:

Imagine working at a place with some of the best and kindest engineers, getting paid to write open source code, and being immersed in one of the most diverse developer communities ever. It sounds too good to be true, doesn't it? When you get a chance to work at npm, you'll be

doing exactly that. Day to day, you'll get to work on challenging problems that push your limits as an engineer. You'll run into problems that'll seem impossible at first, but by the end of your internship will seem like a piece of cake! This internship is one of the most unique opportunities you'll get, so definitely give it a shot and apply!

Faiq worked on our website and search. Even after leaving npm, Faiq continues to be involved and has even [committed to npm core](#). You can expect a similar range of opportunities, depending on your interests.

If you're interested, please send a brief email introducing yourself as well as a PDF or Google doc of your resume to [jobs@npmjs.com](mailto:jobs@npmjs.com) and include the word "internship" in your subject line. The internship is based out of our offices in Oakland, California. You must be part of the graduating class of 2016 or 2017 and legally able to work in the United States; we are not currently able to sponsor visa applications.

## About npm, Inc.

npm is the package manager for JavaScript. The team is small, and growing quickly. If you join us, you will see the company grow through numerous changes, and take a bunch of different roles.

npm's mission is to take Open Source development to entirely new places. When everyone else is adding force, we work to reduce friction.

npm is not a typical product, and we are not a typical early-stage "work hard/play hard" startup. We are responsible adults with diverse backgrounds and interests, who take our careers and our lives seriously. We believe that the best way to iterate towards success is by taking care of ourselves, our families, our users, and one another. We aim for a sustainable approach to work and life, because that is the best way to maximize long-term speed, while retaining clarity of vision. Compassion is our strategy.

[Our offices](#) are in downtown Oakland, California. We offer very competitive salaries, meaningful equity, and generous health, dental and vision benefits. We love it when you represent us at conferences.

---

Last modified June 17, 2016

Found a typo? Send a [pull request!](#)

## npm Private Modules

# Publish, share and install proprietary code easily

[Private modules](#) are ordinary npm packages that only you, and people you select, can view, install, and publish. You publish them in your namespace or your team's namespace, just by giving them a name in package.json:

```
{
  "name": "@myuser/mypackage"
}
```

You publish them with **npm publish**, just like any other package, and you install them by name:

```
npm install @myuser/mypackage
```

Once installed, use them by requiring them by name, just like any package:

```
var mypackage = require('@myuser/mypackage');
```

## Re-use your code

You re-use code between projects. npm and the registry make it really easy to share small modules of useful code with the world. But sometimes the code in that package is private, or sensitive, or just too specific to your needs for you to want to publish it to the public registry. Private packages are great for this.

## Share proprietary code

You work in a team, or you work for clients. You want to be able to easily share your work, with the dependency management and version management that npm provides. By making it easy and granular to select who can see, install and publish packages, [private packages](#) make this easy.

## Security

Need to report a security vulnerability? Please [contact us](#) or email [security@npmjs.com](mailto:security@npmjs.com).

## System Security

Our engineering team is well-versed in security best practices.

Our software is regularly audited by reputable third-party security firms, currently [Lift Security](#).

We maintain a recent, production-ready OS that is regularly patched with the latest security fixes.

Our servers live behind a firewall that only allows expected traffic on limited ports.

Our services are fronted by a CDN that allows for protection from Distributed Denial of Service (DDoS) attacks.

## Security in Transit

All private data exchanged with npm from the command line and via the website is passed over encrypted connections (HTTPS and SSL).

## Physical and Data Security

npm's servers are hosted on Amazon Web Services. Physical security is maximized because nobody knows exactly which physical servers host our virtual ones.

All registry data and binaries are stored in multiple redundant, physically separate locations. All binaries and metadata are backed up to a third-party, off-site location. These backups are encrypted.

Employees of npm Inc. have access to package metadata and binaries for support and debugging purposes. Employees do not have access to the password for your npm account, which is always encrypted.

For more information about how we handle your personal data, you may wish to review our [privacy policy](#).

## Higher Levels of security

For firms interested in greater levels of physical and operational security, [npm Enterprise](#) is a self-hosted version of the npm Registry that allows total control of the operation and policies of the registry.

## Contact Us

If you have further questions or concerns about npm security, please [contact us](#).

## npm Weekly

Find out what we've been working on, thinking about, and talking about every week.

Email Address

subscribe to the weekly

Did you miss one? Check out the [npm weekly archive](#).

---

Last modified November 29, 2015

Found a typo? Send a [pull request!](#)