

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 whatever the repository uses as its default branch.

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 [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`
- `.gitignore`
- `.hg`
- `.npmignore`
- `.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`

If, given the structure of your project, you find `.npmignore` to be a maintenance headache, you might instead try populating the `files` property of `package.json`, which is an array of file or directory names that should be included in your package. Sometimes manually picking which items to allow is easier to manage than building a block list.

Testing whether your `.npmignore` or `files` config works

If you want to double check that your package will include only the files you intend it to when published, you can run the `npm pack` command locally which will generate a tarball in the working directory, the same way it does for publishing.

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