npm-folders

Folder Structures Used by npm

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, it's `%AppData%\npm`. 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 `%AppData%/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.