Synopsis

```
npm link (in package dir)
npm link [<@scope>/]<pkg>[@<version>]

alias: ln
```

Description

This is handy for installing your own stuff, so that you can work on it and test iteratively without having to continually rebuild.

Package linking is a two-step process.

First, `npm link` in a package folder will create a symlink in the global folder `{prefix}/lib/node_modules/<package>` that links to the package where the `npm link` command was executed. It will also link any bins in the package to `{prefix}/bin/{name}`. Note that `npm link` uses the global prefix (see `npm prefix -g` for its value).

Next, in some other location, `npm link package-name` will create a symbolic link from globally-installed `package-name` to `node_modules/` of the current folder.

Note that `package-name` is taken from `package.json`, *not* from the directory name.

The package name can be optionally prefixed with a scope. See `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, if they are included in `bundleDependencies`.

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 redis
```

That is, it first creates a global link, and then links the global installation target into your project's `node_modules` folder.

Note that in this case, you are referring to the directory name, `node-redis`, rather than the package name `redis`.

If your linked package is scoped (see `scope`) your link command must include that scope, e.g.

```
npm link @myorg/privatepackage
```

Caveat

Note that package dependencies linked in this way are *not* saved to `package.json` by default, on the assumption that the intention is to have a link stand in for a regular non-link dependency. Otherwise, for example, if you depend on `redis@^3.0.1`, and ran `npm link redis`, it would replace the `^3.0.1` dependency with `file:../path/to/node-redis`, which you probably don't want! Additionally, other users or developers on your project would run into issues if they do not have their folders set up exactly the same as yours.

If you are adding a *new* dependency as a link, you should add it to the relevant metadata by running `npm install <dep> --package-lock-only`.

If you *want* to save the `file:` reference in your `package.json` and `package-lock.json` files, you can use `npm link <dep> --save` to do so.

Workspace Usage

`npm link <pkg> --workspace <name>` will link the relevant package as a dependency of the specified workspace(s). Note that It may actually be linked into the parent project's `node_modules` folder, if there are no conflicting dependencies.

`npm link --workspace <name>` will create a global link to the specified workspace(s).

Configuration

`save`

- Default: `true` unless when using `npm update` or `npm dedupe` where it defaults to `false`
- Type: Boolean

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

When used with the `npm rm` command, removes the dependency from `package.json`.

Will also prevent writing to `package-lock.json` if set to `false`.

`save-exact`

- Default: false
- Type: Boolean

Dependencies saved to package.json will be configured with an exact version rather than using npm's default semver range operator.

`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 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`

`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.

`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.

`strict-peer-deps`

- Default: false
- Type: Boolean

If set to `true`, and `--legacy-peer-deps` is not set, then *any* conflicting `peerDependencies` will be treated as an install failure, even if npm could reasonably guess the appropriate resolution based on non-peer dependency relationships.

By default, conflicting `peerDependencies` deep in the dependency graph will be resolved using the nearest non-peer dependency specification, even if doing so will result in some packages receiving a peer dependency outside the range set in their package's `peerDependencies` object.

When such and override is performed, a warning is printed, explaining the conflict and the packages involved. If `--strict-peer-deps` is set, then this warning is treated as a failure.

`package-lock`

- Default: true
- Type: Boolean

If set to false, then ignore `package-lock.json` files when installing. This will also prevent *writing* `package-lock.json` if `save` is true.

When package package-locks are disabled, automatic pruning of extraneous modules will also be disabled. To remove extraneous modules with package-locks disabled use `npm prune`.

This configuration does not affect `npm ci`.

`omit`

- Default: 'dev' if the `NODE_ENV` environment variable is set to 'production', otherwise empty.
- Type: "dev", "optional", or "peer" (can be set multiple times)

Dependency types to omit from the installation tree on disk.

Note that these dependencies *are* still resolved and added to the `package-lock.json` or `npm-shrinkwrap.json` file. They are just not physically installed on disk.

If a package type appears in both the `--include` and `--omit` lists, then it will be included.

If the resulting omit list includes `'dev'`, then the `NODE_ENV` environment variable will be set to `'production'` for all lifecycle scripts.

`ignore-scripts`

- Default: false
- Type: Boolean

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

Note that commands explicitly intended to run a particular script, such as `npm start`, `npm stop`, `npm restart`, `npm test`, and `npm run-script` will still run their intended script if `ignore-scripts` is set, but they will *not* run any pre- or post-scripts.

`audit`

- Default: true
- Type: Boolean

When "true" submit audit reports alongside the current npm command to the default registry and all registries configured for scopes. See the documentation for `npm audit` for details on what is submitted.

`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.

`fund`

- Default: true
- Type: Boolean

When "true" displays the message at the end of each `npm install` acknowledging the number of dependencies looking for funding. See `npm fund` for details.

`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`, as well as `pack` and `publish`.

Note: This is NOT honored by other network related commands, eg `dist-tags`, `owner`, etc.

`workspace`

- Default:
- Type: String (can be set multiple times)

Enable running a command in the context of the configured workspaces of the current project while filtering by running only the workspaces defined by this configuration option.

Valid values for the `workspace` config are either:

- Workspace names
- Path to a workspace directory
- Path to a parent workspace directory (will result in selecting all workspaces within that folder)

When set for the `npm init` command, this may be set to the folder of a workspace which does not yet exist, to create the folder and set it up as a brand new workspace within the project.

This value is not exported to the environment for child processes.

`workspaces`

- Default: null
- Type: null or Boolean

Set to true to run the command in the context of **all** configured workspaces.

Explicitly setting this to false will cause commands like `install` to ignore workspaces altogether. When not set explicitly:

- Commands that operate on the `node_modules` tree (install, update, etc.) will link workspaces into the `node_modules` folder. - Commands that do other things (test, exec, publish, etc.) will operate on the root project, *unless* one or more workspaces are specified in the `workspace` config.

This value is not exported to the environment for child processes.

`include-workspace-root`

- Default: false
- Type: Boolean

Include the workspace root when workspaces are enabled for a command.

When false, specifying individual workspaces via the `workspace` config, or all workspaces via the `workspaces` flag, will cause npm to operate only on the specified workspaces, and not on the root project.

See Also

- npm developers
- package.json
- npm install
- npm folders
- npm config

- npmrc