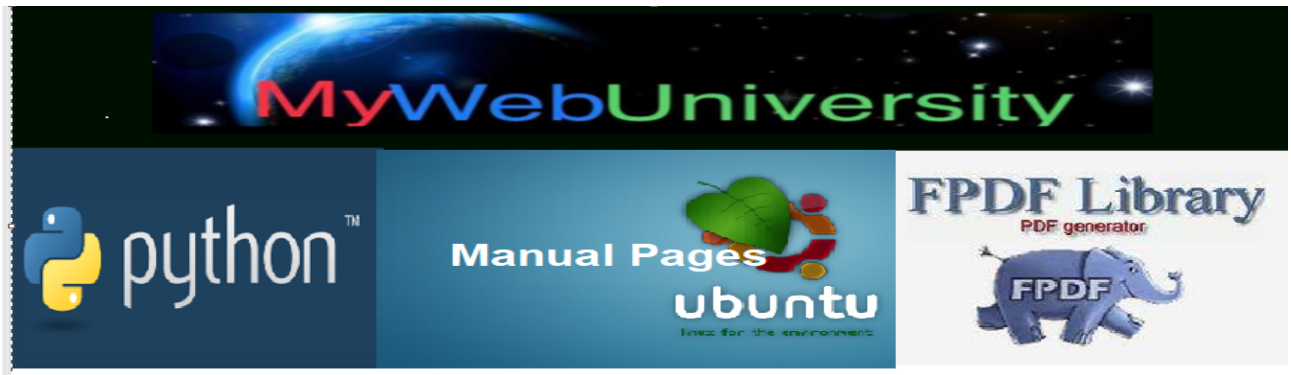

Full credit is given to the above companies including the OS that this PDF file was generated!

Linux Ubuntu 26.04 Manual Pages on command 'npm-link.1'

$ man npm-link.1

NPM-LINK(1) General Commands Manual NPM-LINK(1)

NAME

npm-link

Synopsis

<!-- AUTOGENERATED USAGE DESCRIPTIONS -->

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 with no arguments 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

<!-- AUTOGENERATED CONFIG DESCRIPTIONS -->

See Also

- ? package spec
- ? npm developers
- ? package.json
- ? npm install
- ? npm folders
- ? npm config
- ? npmrc