Table of Contents

npm 핸드북

npm 사용자 설명서입니다.

팩키지와 모듈

팩키지

다음의 어떤 것도 팩키지가 될 수 있습니다.

- a) package.json 파일에 의해 표현되는 프로그램을 포함하고 있는 폴더
- b) (a)를 gzip으로 압축한 파일
- c) (b)에 대한 url
- d) (c)를 통해서 레지스트리에 퍼블리쉬된 `<name>@<version>`
- e) (d)를 가리키는 `<name>@<tag>`
- f) `latest` 로 태구된 `<name>`
- g) (a)로 clone이 되는 git URL

여러분의 팩키지를 공개된 레지스크리에 퍼블리쉬하지 않더라도 npm을 사용함으로써 얻을 수 있는 많은 장점이 있습니다.

- 그냥 node 프로그램을 작성할 수 도 있으며
- 이 프로그램을 압축해서 다른 곳에 쉽게 설치할 수 있습니다.

모듈

Node.js 프로그램내에서 `require()` 로 로드할 수 있는 것을 모듈이라고 합니다. 다음의 예 모두가 모듈로 로드될 수 있습니다.

- `main` 필드를 가진 `package.json` 파일을 포함하고 있는 폴더
- `index.js` 파일을 포함하고 있는 폴더
- JavaScript 파일

대부분의 npm 팩키지는 모듈

일반적으로 Node.js 프로그램에서 사용된느 npm 팩키지들은 `require` 를 통해서 로드되므로 모듈이라고 할 수 있습니다. 하지만 npm 팩키지가 반드시 모듈이어야할 필요는 없습니다.

실행가능한 명령행 인터페이스를 포함한ㄴ `cli` 와 같은 팩키지는 Node.js 프로그램에서 사용할 수 있도록 하는 `main` 필드가 없습니다. 따라서 이 팩키지는 모듈이 아닙니다.

대부분의 팩키지 (Node 프로그램이라면)는 많은 모듈을 포함하고 있습니다.

Node 프로그램의 문맥상, `module` 은 파일에서 로드되는 것을 뜻합니다.

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

Node.js와 npm 생태계에서의 파일과 디렉터리 이름

`package.json` 파일은 팩키지를 정의합니다.

`node_modules` 폴더는 Node.js가 모듈을 찾는 장소입니다.

예를 들어 node_modules/foo.js라는 파일을 생성하고 프로그램에서 모듈을 로드하기 위해서 `var f = require('foo.js')` 라고 하면 `foot.js` 에는 package.json 파일이 없으므로 "package"가 아닙니다.

이와 다르게, `index.js` 이 없고 `package.json` 내에 `main` 필드가 없는 팩키지를 만들었다면 이 팩키지는 모듈이 아닙니다. 따라서 `node_modules` 에 설치된다 하더라도 `require()` 로 로드될 수 없으므로 모듈이 아닙니다.

npm-config

More than you probably want to know about npm config

Description

npm은 설정 값들을 아래의 우선순위별로 설명한 장소로부터 읽어옵니다.

명령행 플래그

명령행에 `--foo bar` 를 지정하면 **foo** 설정 파라미터에 **"bar"**를 지정합니다. **--** 인자는 명령행 인터페이스 파서에게 플래그로 읽지 않도록 알려줍니다. 값이 지정되지 않은 **--flag** 파라미터는 **true**로 해석됩니다.

환경변수

npm_config_로 시작하는 모든 환경변수는 설정 파라미터로 해석됩니다. 예를 들어, **npm_config_foo=bar**를 환경변수로 설정하면 **foo** 파라미터에 **bar** 값이 지정됩니다. 값을 지정하지 않은 환경변수 설정 값은 **true**로 해석됩니다. 환경변수의 대소문자는 구별하지않으므로 **NPM_CONFIG_FOO=bar**는 동일한 효과를 냅니다.

npmrc 파일

네 가지 파일이 관련되어 있습니다.

- 프로젝트별 설정 파일 (\/path\/to\/my\/project\/.npmrc)
- 사용자별 설정 파일 (~\/.npmrc)
- 전역 설정 파일 ($PREFIX\/etc\/npmrc)
- npm 내장 설정 파일 (\/path\/to\/npm\/npmrc)

npmrc 파일에 대한 자세한 사항은 [Configuring npm] 장을 참고하기 바랍니다.

디폴트 설정

`npm config ls -l` 을 실행하면 npm 설정 파라미터들을 확인해 볼 수 있으며 별도로 지정하지 않은 항목에 대한 디폴트 값들도 확인할 수 있습니다.

명령행 파라미터의 단축형

다음과 같은 단축형 파라미터가 있습니다.

- **-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** 와 **la** 명령: **ls --long**

지정된 설정 파라미터가 명확하게 알려진 설정 파라미터로 해석될 수 있는 경우에는 해당 설정 파라미터로 확장됩니다. 예를 들어

```
npm ls --par
# 아래와 같습니다.
npm ls --parseable
```

단일문자인 단축형 파라미터를 다중으로 함께 묶어서 사용할 수 있습니다.

```
npm ls -gpld
# 아래와 같습니다.
npm ls --global --parseable --long --loglevel info
```

팩키지별 설정

스크립트를 실행할 때 package.json의 "config"의 키가 환경변수에 설정된 값들을 덮어쓰게 됩니다. 예를 들어 package.json에 다음과 같이 되어 있고...

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

server.js가 아래와 같다면

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

이 상황에서 사용자는 다음과 같이 설정사항을 변경할 수 있습니다.

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

Config 설정

access

- 디폴트: restricted
- 타입: Access

스코프가 지정된 팩키지를 퍼블리싱할때, 액세스 레벨은 디폴트로 **restriected**가 됩니다. 만약 팩키지를 공개적으로 개방하려면 (인스톨가능하도록 하려면) **--access=public**으로 설정해야합니다. **access**에 지정할 수 있는 값은 **public**과 **restricted**입니다. 스코프가 지정되지 않은 팩키지는 항상 **public** 수준의 액세스가됩니다.

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 **npat** 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 elminate 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-attribs

- 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 bundledDependencies 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.

npm-scripts

How npm handles the "scripts" field

Description

아래와 같은 package.json 파일에서 "scripts" 프로퍼티를 사용할 수 있습니다.

- prepublish: 팩키지를 퍼블리쉬하기전에 실행됩니다. (또한 로컬에서 다른 인자 없이 **npm install**를 실행할 때에도 적용됩니다.)
- publish, postpublish: 팩키지를 퍼블리쉬한 후에 실행됩니다.
- preinstall: 팩키지를 인스톨하기 전에 실행됩니다.
- install, postinstall: 팩키지를 인스톨한 후에 실행됩니다.
- preuninstall, uninstall: 팩키지를 언인스톨하기 전에 실행됩니다.
- postuninstall: 팩키지를 언이스톨한 후에 실행됩니다.
- 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. 주의: **npm restart** 시에 restart 스크립트를 지정하지 않으면 stop과 start 스크립트가 실행됩니다.

일반적인 활용

팩키지를 사용하기 전에 팩키지를 어떤 작업을 수행해야하는데 이 작업이 운영 시스템이나 대상 시스템의 아키텍쳐에 의존적이지 않다면 **prepublish** 스크립트를 사용합니다. 예를 들어 다음과 같은 경우입니다.

- CoffeeScript 소스코드를 JavaScript로 컴파일
- JavaScript 소스코드를 minified된 버전으로 생성
- 팩키지에서 사용해야하는 리모트 리소스를 받아오기

이런 작업을 **prepublish**에 기술하게 되면 변경이 발생할 때 이 부분만 수정하면 되므로 복잡도를 낮출 수 있습니다.

- **coffee-script**를 **devDependency**에만 의존성을 부여하므로 팩키지의 사용자는 추가로 설치할 필요가 없습니다.
- minifier를 팩키지에 포함할 필요가 없으므로 팩키지의 크기를 줄일 수 있습니다.

- 사용자가 **curl**이나 **wget**을 실행할 필요가 없습니다.

디폴트 값

팩키지의 내용에 따라 npm은 몇 가지 디폴트 스크립트 값을 가지게 됩니다.

- "start": "node server.js"

만약 팩키지의 최상위 경로에 server.js가 존재하면 npm은 **start** 명령어로 **start server.js**를 디폴트로 설정됩니다.

- "install": "node-gyp rebuild"

만약 팩키지 최상위 경로에 binding.gyp 파일이 존재하면 npm은 **install** 명령어로 node-gyp를 사용하여 컴파일하게 됩니다.

사용자

npm을 root 권한으로 실행하면 사용자 계정의 uid나 **user** 설정에 지정된 uid로 변경하게 되는데 이 값은 디폴트로 **nobody**입니다. root 권한으로 스크립트를 실행하려면 **unsafe-perm** 플래그를 설정해야합니다.

환경변수

팩키지 스크립트는 npm의 설정과 프로세스의 현재 상태에 따라 다양한 정보들이 환경 변수로 받아 올 수 있습니다.

path

실행 가능한 스크립트를 정의하는 모듈에 따라서 이들 실행 파일들을 실행할 수 있도록 **PATH** 에 추가됩니다. 따라서 만약 package.json에 다음과 같이 설저되어 있다면

```
{ "name" : "foo"
, "dependencies" : { "bar" : "0.1.x" }
, "scripts": { "start" : "bar ./test" } }
```

npm install을 통해서 **node_modules/.bin** 디렉터리에 익스포트된 **bar** 스키립트를 실행하기 위해서 **npm start**를 실행할 수 있습니다.

package.json vars

package.json 필드들은 **npm_package_** 프리픽스를 추적합니다. 예를 들어 package.json 파일 내에 `{"name":"foo", "version":"1.2.5"}` 로 되어 있다면, 팩키지 스크립트에서는 **npm_package_name** 환경 변수를 통해서 **foot**값을 참조할 수 있으며 **npm_package_version** 은 "1.2.5"로 설정되게 됩니다.

configuration

설정 파라미터들은 **npm_config_**로 시작하는 환경 변수로 들어갑니다. 예를 들어 **root** 설정은 **npm_config_root** 환경 변수를 통해서 확인할 수 있습니다.

package.json의 config 객체

<name>[@version]: 형태의 package.json의 "config" 키이 값은 환경변수의 값에 의해 덮어 쓰여집니다. 예를 들어 package.json이 아래와 같고

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

server.js가 아래와 같다면

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

사용자는 아래와 같이 함으로써 변경할 수 있습니다.

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

current lifecycle event

마지막으로 **npm_lifecycle_event** 환경 변수는 실행 중인 cycle 단계에 따라 변경됩니다. 따라서 하나의 스크립트에서 현재 실행되는 상태에 따라 다른 일을 하도록 설정할 수 있습니다.

package.json에 ``으로 되어 있다면 스크립트 내에서는 아래와 같이 보입니다.

```
process.env.npm_package_scripts_install === "foo.js"
```

EXAMPLES

```
{ "scripts" :
 { "install" : "scripts/install.js"
 , "postinstall" : "scripts/install.js"
 , "uninstall" : "scripts/uninstall.js"
 }
}
```

```
{ "scripts" :
 { "preinstall" : "./configure"
 , "install" : "make && make install"
 , "test" : "make test"
 }
}
```

EXITING

스크립트는 **sh**에 파라미터로 전달되어 실행됩니다.

만약 스크립트가 0이 아닌 코드 값으로 종료되면, 프로세스가 강제 종료(*abort*)됩니다.

스크립트 자체는 javascript 또는 nodejs 프로그램일 필요는 없으며 그냥 실행 가능한 파일이면 됩니다.

HOOK SCRIPTS

모든 팩키지에 대해서 특정 라이프사이클 이벤트에 특정 스크립트를 실행하고 싶다면 후크 스크립트를 사용할 수 있습니다.

실행가능한 파일을 `node_modules/.hooks/{eventname}` 에 두면, 모든 팩키지에 대해서 해당 라이프사이클로 진입할 때 실행됩니다.

후크 스크립트는 package.json 스크립트가 실행되는 것과 동일한 방법으로 별도의 자식 프로세스로 위에서 설명한 환경상에서 실행됩니다.

BEST PRACTICE

- 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.

npm-access

퍼블리쉬된 팩키지의 액세스 레벨을 설정합니다.

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

프라이빗 팩키지에 대한 접근 권한을 설정합니다.

서브 명령어를 수행할 때, 팩키지 이름을 지정하지 않으면 `npm access` 는 현재 작업 디렉터리에 포함된 팩키지들에 대해서 적용됩니다.

- public / restricted : 공개된 팩키지인지 제한된 팩키지인지 설정합니다.
- grant / revoke : 사용자와 팀이 팩키지에 대해 읽기만 허용되는지 읽고 쓰기가 가능한지 권한을 추가하거나 제거합니다.
- ls-packages : 사용자나 팀이 접근할 수 있는 모든 팩키지를 접근 권한에 따라 나열합니다. 전체 레지스트리를 나열하지 않도록 읽기 전용 퍼블릭 팩키지는 목록에서 제외됩니다.
- ls-collaborators : 팩키지에 대한 모든 접근 권한을 나열합니다. 최소한 읽기 권한을 가진 팩키지의 퍼미션을 나열하게 됩니다. 만약 `<user>` 가 지정되면, 이 사용자가 속한 팀만 필터됩니다.
- edit : `$EDITOR` 를 사용하여 팩키지의 접근 권한을 설정합니다.

DETAILS

`npm access` 는 항상 현재 레지스트리에 직접접으로 작용하는데 명령행에서 `--registry=<registry url>` 을 통해서 설정할 수 있습니다.

스코프화 되지 않은 팩키지(*unscoped packages*)는 항상 퍼블릭(*public*)입니다.

스코프 팩키지(*scoped packages*)는 디폴트로 제한된 상태(*restricted*)입니다. `npm publish --access=public` 을 사용하여 퍼블릭으로 공개할 수도 있습니다. 또한 최초의 퍼블리쉬 이후에 `npm access public` 액세스 설정을 변경할 수 있습니다.

팩키지 액세스 설정을 변경하려면 권한이 있어야합니다.

- 스코프 또는 스코프화 되지 않은 팩키지의 소유자
- 스코프를 소유한 팀의 멤버
- 팩키지에 읽기-쓰기 권한이 허가된 사용자 이거나 소유자 또는 팀의 멤버.

유료 사용자가 아니면 `--access=publish` 를 지정하지 않고 스코프 팩키지를 배포하려고 할 때 HTTP 402 상태코드의 오류가 발생하게 됩니다.

npm-adduser

레지스트리 사용자 계정 추가

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.

npm-bin

npm의 bin 폴더를 출력

Synopsis

```
npm bin [-g|--global]
```

Description

npm이 실행 파일을 설치할 폴더의 경로를 출력합니다.

npm-bugs

버그 리포팅

Synopsis

```
npm bugs [<pkgname>]
```

Description

팩키지의 버그 트랙커 URL 주소를 가능한 추측하여 `--browser` 설정 파라미터에 지정된 방법으로 열어줍니다. 팩키지 이름이 지정되지 않으면 현재 폴더의 `package.json` 을 찾아서 `name` 프로퍼티를 대신 사용하게 됩니다.

Configuration

browser

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

`npm bugs` 명령으로 웹사이트를 열기위해 호출되는 브라우저

registry

- Default: https:\/\/registry.npmjs.org\/
- Type: url

npm 팩키지 레지스트리의 기본 URL

npm-build

팩키지 빌드

Synopsis

```
npm build [<package-folder>]
```

- `<package-folder>` : `package.json` 파일을 포함하고 있는 팩키지의 최상위 폴더.

Description

`npm link` 와 `npm install` 에 의해서 호출되는 기반 명령어입니다.

일반적으로 인스톨 과정에서 호출되어야 하지만 직접 실행하려면 `npm run-script build` 를 입력합니다.

npm-bundle

제거되었음.

Description

디폴트로 팩키지를 로컬에 설치하게 되어 더 이상 필요 없는 관계로 `npm bundle` 명령은 1.0 버전에서 제거되었습니다.

이전에 `npm bundle` 이 하던 일을 하려면 `npm install` 을 대신 사용하면 됩니다.

npm-cache

팩키지 캐시 관리

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

npm 캐시 폴더에 팩키지를 추가하고 리스트를 출력하고 캐시를 비우는 작업을 합니다.

- add: 지정된 팩키지를 로컬 캐시에 추가합니다. 이 명령어는 주로 npm이 내부적으로 사용하기 위해서 만들어 졌습니다. 하지만 로컬 캐시애 명시적으로 데이터를 추가할 수 있는 방법을 제공합니다.
- ls: 캐시에 저장된 디에터를 출력합니다. 인자는 출력할 캐시 폴더내의 경로입니다. `find` 프로그램처럼 동작하며 `depth` 설정으로 범위를 제한할 수 있습니다.
- clean: 캐시 폴더의 데이터를 삭제합니다. 인자가 지정되면 인자를 삭제해야할 하위 경로로 간주합니다. 인자가 없이 실행하면 전체 캐시를 삭제합니다.

DETAILS

npm은 `npm config get cache` 에 지지ㅓㅇ된 디렉터리에 캐시 데이터를 저장하고 있습니다. 캐시에 정장된 개별 팩키지는 `{cache}/{name}/{version}` 에 세 가지 정보를 저장합니다.

- .../package/package.json : npm이 참고하는 package.json 파일
- .../package.tgz : 해당 버전의 패키지 압축 파일

추가로 레지스트리에 대한 요청을 할 때마다 ETag와 요청 데이터를 저장하기 위해 `{cache}/{hostname}/{path}/.cache.json` 파일이 만들어집니다.

`search` , `view` , `completion` 과 같은 부수적인 레지스트리에 대한 요청은 최소한의 타이아웃 시간이 지정됩니다. `.cache.json` 파일의 생성시간이 지정된 타임아웃 이전의 시간이라면 레지스틀에 대한 HTTP 요청은 생성되지 않습니다.

Configuration

cache

디폴트: Posix 시스템에서는 `~/.npm` , Windows 시스템에서는 `%AppData%/npm-cache`

캐시 폴더의 최상위 경로

npm-completion

탭을 사용한 자동 완성

Synopsis

```
source < (npm completion)
```

Description

모든 npm 명령어의 탭 완성기능을 가능하게 합니다.

현재 쉘에서 `source < (npm completion)` 을 입력하면 탭완성 기능을 로드합니다. 이 내용을 ~/.bashrc나 ~/.zshrc에 추가하면 언제나 자동 완성기능을 사용할 수 있게됩니다.

```
npm completion >> ~/.bashrc
npm completion >> ~/.zshrc
```

npm 완성의 출력물을 `/usr/local/etc/bash_completion.d/npm` 와 같은 파일에 저장하여 시스템에서 읽어 들이도록 할 수 있습니다.

`COMP_CWORD` , `COMP_LINE` , `COMP_POINT` 를 환경변수로 정의할 때, `npm completion` 은 `plumbing mode` 로 동작하여 명령행 인자에 기반하여 완성기능이 출력됩니다.

npm-config

Manage the npm configuration files

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

npm의 설정 항목들을 명령행, 환경변수, `npmrc` 파일에서 가져옵니다. 몇 가지 경우에는 `package.json` 파일에서 가져오게 됩니다.

npmrc 파일에 대한 정보는 npmrc를 참고하기 바랍니다.

관련된 동작방식에 대해서는 Configruing npm 챕터를 참조하기 바랍니다.

`npm config` 명령으로 사용자 및 글로벌 npmrc 파일의 내용을 수정할 수 있습니다.

Sub-commands

다음에 설명하는 서브 명령어들을 지원합니다.

set

```
npm config set key value
```

주어진 키로 값을 설정합니다.

값을 생략하면 "true"를 값으로 사용합니다.

get

```
npm config get key
```

설정 값을 stdout으로 출력합니다.

list

```
npm config list
```

모든 설정값을 출력합니다.

delete

```
npm config delete key
```

모든 설정 파일에서 키에 해당하는 항목을 삭제합니다.

edit

```
npm config edit
```

에디터로 설정 파일을 열어줍니다. 글로벌 설정 파일을 수정하려면 `--global` 플래그를 사용해야 합니다.

npm-dedupe

중복 항목을 제거

Synopsis

```
npm dedupe
npm ddp
```

Description

로컬 팩키지 트리를 검색하여 트리에 트리에 걸쳐있는 의존성 패키지를 정리하여 여러 팩키지에서 공통으로 의존하는 것들을 효과적으로 공유하도록 만듭니다.

아래와 같은 의존성 그래를 예로 들면,

```
a
+-- b <-- depends on c@1.0.x
| `-- c@1.0.3
`-- d <-- depends on c@~1.0.9
 `-- c@1.0.10
```

이 경우에 `npm dedupe` 는 아래와 같이 트리를 변경합니다.

```
a
+-- b
+-- d
`-- c@1.0.10
```

Node이 모듈을 검색하는 계층적인 속성 때문에, b와 d는 의존하고 있는 c 팩키지를 트리의 최상위 레벨에서 찾을 수 있게 됩니다.

중복을 찾는 알고리즘은 트리를 순회하면서 중복된 의존성이 없는 경우에도 각각의 의존 팩키지를 가능한 최상위로 옮기려고 합니다. 이 결과로 평탄하면서 중복성이 없어진 트리가 만들어 집니다.

만약 알맞은 버전의 팩키지가 트리내 적절한 위치에 이미 존재하는 경우라면, 팩키지를 손대지 않고 다른 중복된 팩키지만 삭제합니다.

명령행의 인자는 무시되면 `dedupe` 는 항상 전체 트리에 대해서 작용합니다.

이 작업은 의존성 트리의 변경만하며, 새로운 모듈을 설치하지 않는다는 점에 주의해야합니다.

npm-deprecate

팩키지의 특정 버전을 디프리케이트

Synopsis

```
npm deprecate <pkg>[@<version>] <message>
```

Description

이 명령어로 npm 레지스트리에 있는 팩키지 엔트리를 업데이트하여 설치하려고 시도하는 사용자에게 디프리케이션 경고를 발생시킵니다.

특정 버전 또는 일부 버전 범위에 적용할 수 있습니다. 이 경우 다음과 같이 합니다.

```
npm deprecate my-thing@"< 0.2.3" "critical bug fixed in v0.2.3"
```

뭔가를 디프리케이트 시키려면 해당 패키지의 소유권자이어야합니다. `owner` 와 `adduser` 에 관한 설명을 참조하기 바랍니다.

디프리케이트된 팩키지를 되돌리려면 빈 문자열 (`""`)를 `message` 인자에 지정하면 됩니다.

npm-dist-tag

팩키지의 배포 태그 수정

Synopsis

```
npm dist-tag add <pkg>@<version> [<tag>]
npm dist-tag rm <pkg> <tag>
npm dist-tag ls [<pkg>]
```

Description

팩키지의 배포 태그를 추가, 삭제, 출력합니다.

- add : 특정 버전의 팩키지에 특정 태그를 답니다.
- rm : 팩키지에서 더 이상 사용되지 않는 태그를 제거합니다.
- ls : 팩키지의 모든 dist-tag를 출력합니다.

태그는 팩키지를 설치할 때 특정 버전 숫자 대신에 팩키지의 버전을 참조하기 위해서 사용될 수 있습니다.

```
npm install <name>@<tag>
```

의존 팩키지를 설치할 때, 원하는 태그 버전을 지정할 수 있습니다.

```
npm install --tag <tag>
```

이 명령은 `npm dedupe` 에도 적용됩니다.

팩키지를 퍼블리싱할 때 `--tag` 옵션을 사용하여 `npm publish --tag=beta` 와 같이 하지 않으면 `latest` 태그가 지정됩니다.

PURPOSE

태그를 버전 숫자를 대신하는 별명 정도로 사용할 수 있습니다.

예를 들어, 개발과정에서 프로젝트에 여러 개의 스트림을 선택할 수 있으며 각 스트림에서 각기 다른 태그를 사용할 수 있습니다. 예) `stable` , `beta` , `dev` , `cnary`

디폴트로, npm은 현재 버전의 팩키지를 지정하기위해서 `latest` 태그를 사용하며 (`@<version>` 또는 `@<tag>` 을 지정하지 않고) `npm install <pkg>` 를 하면 `latest` 태그를 설치합니다. 일반적인 프로젝트에서는 안정적인 릴리즈 버전으로 `latest` 태그만 사용하며 프리릴리즈와 같은 안정적이지 않은 버전에는 다른 태그를 사용합니다.

어떤 프로젝트에서는 `next` 태그를 앞으로 릴리즈될 버전의 식별자로 사용하기도 합니다.

npm 자체에서 특별히 구분하는 태그는 디폴트 `latest` 이 외에는 없습니다.

CAVEATS

이 명령은 이전에는 `npm tag` 였었는데 지금은 새로운 태그를 생성하는 용도로 별도의 문법을 가지고 있습니다.

태그는 버전 숫자와 `npm install <pkg>@<version>, npm install <pkg>@<tag>` 와 같이 동일한 위치를 사용해야하므로 네임스페이스를 공유해야합니다.

`v1.4` 와 같이 적법한 semver 범위로 인식되어 `>=1.4.0 <1.5.0` 와 같은 의미로 해석될 수 있는 태그는 태그로 사용될 수 없습니다.

semver과의 충돌을 피할 수 있는 가장 간단한 방법은 태그를 숫자나 `v` 문자로 시작하지 않도록 하는 것입니다.

npm-docs

팩키지 문서화

Synopsis

```
npm docs [<pkgname> [<pkgname> ...]]
npm docs .
npm home [<pkgname> [<pkgname> ...]]
npm home .
```

Description

이 명령은 팩키지의 도큐먼트 URL의 위치를 추측하여 `--browser` 설정 파라미터를 사용하여 열어줍니다. 하나 이상의 팩키지 이름을 한 번에 사용할 수 있으며, 팩키지 이름을 지정하지 않으면 대신 현재 폴더에서 `package.json` 을 찾아서 `name` 프로퍼티를 이용합니다.

Configuration

browser

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

`npm docs` 명령에서 웹사이트를 열기위해 호출될 브라우저

registry

- Default: http://registry.npmjs.org/
- Type: url

npm 팩키지 레지스트리의 베이스 URL

npm-edit

실치된 팩키지를 수정

Synopsis

```
npm edit <pkg>[@<version>]
```

Description

npm 설정에서 `editor` 에 지정된 에디터를 사용하여 팩키지 폴더를 열어줍니다.

수정이 완료된 이후에 팩키지는 다시 빌드되어 수정사항이 적용됩니다.

예를 들어 `npm install connect` 로 connect를 설치하고 `npm edit connect` 를 통해서 로컬에 설치된 복사본을 수정할 수 있습니다.

Configuration

editor

- Default: `EDITOR` 환경 변수가 설정되어 있다면 이 값이 사용되며, 그렇지 않은 경우에 Posix 시스템에서는 `vi` , Windows 시스템에서는 `notepad`
- Type: path

이 설정 값은 `npm edit` 와 `npm config edit` 에서 사용됩니다.

npm-explore

설치된 팩키지를 탐색

Synopsis

```
npm explore <pkg> [ -- <cmd>]
```

Description

지정된 설치 팩키지의 디렉터리에서 서브 쉘을 실행시킵니다.

명령어가 지정되면, 서브 쉘상에서 명령어를 실행한 후 즉시 쉘을 종료합니다.

`node_modules` 폴더 내에 있는 git 서브 모듈내에서 편리하게 사용할 수 있습니다.

```
npm explore some-dependency -- git pull origin master
```

이후에 팩키지가 자동으로 다시 빌드되지 않는 다는 점에 유의해야합니다. `npm rebuild <pkg>` 를 사용해서 빌드해야합니다.

Configuration

shell

- Default: SHELL 환경변수, 또는 Posix상에서는 "bash", Windows상에서는 "cmd"
- Type: path

`npm explore` 명령에서 실행할 쉘

npm-help

npm 도움말

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.

npm-help-search

npm 도움말 무서를 검색

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.

npm-init

paackage.json 파일을 대화형으로 생성

Synopsis

```
npm init [-f|--force|-y|--yes]
```

Description

사용자에게 몇 가지 질문을 하고 대답에 따라 package.json을 생성합니다.

사용자가 대답한 응답에 따라 적절한 값을을 미리 설정한 package.json을 생성합니다.

만약 package.json 파일이 이미 존재하면, 이 파일을 먼저 읽은 후에 여기에 있는 값들을 디폴트 값으로 사용하게 됩니다.

특별한 이유가 없는한 package.json의 원래 값을 유지한채 새로운 값을 추가하려고 시도합니다.

`-f`, `--force`, `-y`, `--yes` 옵션을 사용하면, 사용자에게 어떠한 질문도 없이 디폴트 값만 사용하여 package.json 파일을 생성합니다.

Configuration

scope

- Default: none
- Type: String

생성하려는 새로운 모듈의 스코프를 지정합니다.

npm-install

팩키지 설치

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

팩키지와 이 패키지가 의존하고 있는 팩키지를 함께 설치합니다. 만약 팩키지에 shrinkwrap 파일이 있으면, 의존 팩키지의 설치는 이것에 의해 결정됩니다. npm-shrinkwrap을 참고.

팩키지라 함은 다음과 같습니다.

- a) package.json 파일에 의해 표현되는 프로그램을 포함하고 있는 폴더
- b) (a)를 gzip으로 압축한 파일
- c) (b)에 대한 url
- d) (c)를 통해서 레지스트리에 퍼블리쉬된 `<name>@
- e) (d)를 가리키는 `<name>@<tag>`
- f) `latest`로 태그된 `<name>`
- g) (a)로 clone이 되는 git URL

여러분의 팩키지를 공개된 레지스크리에 퍼블리쉬하지 않더라도 npm을 사용함으로써 얻을 수 있는 많은 장점이 있습니다. (a) 그냥 node 프로그램을 작성할 수 도 있으며 (b) 이 프로그램을 압축해서 다른 곳에 쉽게 설치할 수 있습니다.

- **`npm install`** (팩키지 디렉터리 내에서, 인자가 없이)

로컬 node_modules 폴더에 의존 팩키지를 설치합니다.

(`-g` 나 `--global` 옵션을 지정한 경우) 글로벌 모드에서는 현재 팩키지 컨텍스트(예: 현재 작업 디렉터리)를 글로벌 팩키지로 설치합니다.

디폴트로, `npm install` 은 package.json에서 dependencies에 나열된 모든 모듈들을 설치합니다.

`--production` 플래그를 주면 (또는 `NODE_ENV` 환경 변수가 `production` 으로 설정되어 있으면), npm은 `devDependencies` 에 나열된 모듈들은 설치하지 않습니다.

- **`npm install <folder>`**

파일 시스템 상의 해당 폴더에 들어 있는 팩키지를 설치합니다.

- **`npm install <tarball file>`**

파일 시스템 상의 해당 팩키지를 설치합니다. 주의: 개발 디렉터리를 npm 최상위 경로에 단순히 링크하려고 하면 `npm link` 명령을 사용하면 됩니다.

```
npm install ./package.tgz
```

- **`npm install <tarball url>`**

압축 파일을 url에서 내려받아서 설치합니다. "http://"나 "https://"로 옵션이 시작하는지에 따라 이 옵션을 다른 옵션과 구분합니다.

```
npm install https://github.com/indexzero/forever/tarball/v0.5.6
```

- **`npm install [<@scope>/]<name> [-S|--save|-D|--save-dev|-O|--save-optional]`**

`<name>@<tag>` 를 설치합니다. 여기서 `<tag>` 는 "tag" 설정을 따릅니다. (npm-config를 참고, 디폴트 tag값은 `latest`)

대부분의 경우 최신 버전의 모듈을 설치하 것입니다.

```
npm install sax
```

`npm install` 은 팩키지 버전을 package.json 파일에 저장하거나 업데이트하는 3가지 배타적인 옵션 플래그를 지원합니다.

- `-S` , `--save` : 팩키지를 `dependencies` 에 추가합니다.
- `-D` , `--save-dev` : 팩키지를 `devDependecies` 에 추가합니다.
- `-O` , `--save-optional` : 팩키지를 `optionalDependencies` 에 추가합니다.

위의 세 가지 중 하나를 사용해서 의존성 정보를 package.json에 삽입할 때, 추가적인 옵셔널 플래그가 지원됩니다.

- `-E` , `--save-exact` : 의존성 팩키지의 정보를 설정할 때 npm의 디폴트 semver 버전 연산자 대신에 구체적인 버전 정보를 사용합니다.

더 나아가, `npm-shrinkwrap.json` 파일이 존재하면 이 파일도 같이 업데이트합니다.

`<scope>` 는 옵셔널입니다. 지정된 스코프에 연관된 팩키지를 레지스트리에서 다운로드합니다. 주어진 스코프에 연관된 레지스트리가 없다면 디폴트 레지스트로 가정합니다.

@-기호를 스코프 이름에 포함하지 않으면 npm은 이 이름을 GitHub 레포지토리로 인식합니다. 스코프 이름은 슬래쉬로 끝나야합니다.

```
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
```

주의: `<name>` 인 파일이나 폴더가 현재 작업 디렉터리에 존재하면 이 것을 설치하려고 시도하게 되며 이 파일이나 폴더가 유효하지 않은 경우에만 이름으로 팩키지를 가져오게 됩니다.

- **`npm install [<@scope>/]<name>@<tag>`**

지정된 태그로 참조되는 팩키지의 버전을 설치합니다. 레지스트리에 해당 태그가 존재하지 않으면 실패하게됩니다.

```
npm install sax@latest
npm install @myorg/mypackage@latest
```

- **`npm install [<@scope>/]<name>@<version>`**

지정된 버전의 팩키지를 설치합니다. 레지스트리에 해당 버전이 존재하지 않으면 실패하게됩니다.

```
npm install sax@0.1.1
npm install @myorg/mypackage@1.5.0
```

- **`npm install [<@scope>/]<name>@<version range>`**

지정된 범위에 해당되는 버전의 패키지가 설치됩니다. package.json에서 설명한 `dependecies` 를 해석하는 규칙이 동일하게 적용됩니다.

버전 범위를 따옴표로 감싸서 쉘에서 하나의 인자로 인식되도록 해야합니다.

```
npm install sax@">=0.1.0 <0.2.0"
npm install @myorg/privatepackage@">=0.1.0 <0.2.0"
```

- **`npm install <git remote url>`**

git 프로바이더에 호스팅된 팩키지를 git clone을 통해서 설치합니다. https(github의 git)를 통해서 먼저 시도하고 실패하는 경우 ssh를 통해서 시도하게 됩니다.

```
<protocol>://[<user>[:<password>]@]<hostname>[:<port>][:][/]<path>[#<commit-ish>]
```

`<protocol>` 은 `git` , `git+ssh` , `git+http` , `git+https` , `git+file` 중의 하나이어야합니다.

`<commit-ish>` 가 명시되지 않으면, `master` 를 사용하게 됩니다.

다음과 같은 git 환경변수를 npm에서 인식하여 git을 실행할 때 환경변수로 설정하게 됩니다.

- `GIT_ASKPASS`
- `GIT_PROXY_COMMAND`
- `GIT_SSH`
- `GIT_SSH_COMMAND`
- `GIT_SSL_CAINFO`
- `GIT_SSL_NO_VERIFY`

```
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>]`**

git을 사용하여 github의 팩키지를 설치합니다. `commit-ish` 가 명시되지 않으면 `master` 를 사용합니다.

```
npm install mygithubuser/myproject
npm install github:mygithubuser/myproject
```

- **`npm install gist:[<githubname>/]<gistID>[#<commit-ish>]`**

git을 사용하여 `https://gist.github.com/gistID` 의 팩키지를 설치합니다. gist에 연결된 GitHub 사용자명은 선택사항이며 `-S` 나 `--save` 가 사용되면 package.json에 저장되지 않습니다.

```
npm install gist:101a11beef
```

- **`npm install bitbucket:<bitbucketname>/<bitbucketrepo>[#<commit-ish>]`**

- `npm install gitlab:<gitlabname>/<gitlabrepo>[#<commit-ish>]`

복수의 인자를 복합적으로 사용할 수 있을 뿐만 아니라 복수의 인자 타입을 사용할 수도 있습니다.

```
npm install sax@">=0.1.0 <0.2.0" bench supervisor
```

`--tag` 인자는 모든 지정된 설치 타겟에 사용될 수 있습니다. 만약 주어진 이름의 태그가 존재하면 태그된 버전이 새로운 버전보다 우선해서 선택됩니다.

`--dry-run` 인자는 실제로는 설치하지 않으면서 설치과정에서의 결과를 출력합니다.

`-f` 또는 `--force` 인자는 로컬에 복사본이 존재하더라도 원격의 리소스를 다운로드하도록 강제합니다.

```
npm install sax --force
```

`-g` 또는 `--global` 인자는 팩키지를 글로벌로 설치합니다.

`--global-style` 은 npm이 팩키지를 로컬 `node_modules` 에 설치하면서 글로벌 `node_moudles` 에 적용되는 동일한 레이아웃을 사용하도록 합니다. 직접적인 의존성 팩키지만 `node_modules` 에 나타나며 이 팩키지들이 의존하는 것들은 모두 평탄화(*flatterned*)됩니다.

`--legacy-bundling` npm 1.4 이전 버전의 팩키지를 설치하도록 만듭니다.

`--link` 로컬 공간에 글로벌 설치본에 대한 링크를 만듭니다.

`--no-bin-links` 인자는 팩키지에 포함되어 있는 바이너리에 대한 심볼릭 링크를 생성하지 않도록 합니다.

`--no-optional` 옵셔널 의존성 팩키지를 설치하지 않도록 합니다.

`--no-shrinkwrap` 는 shrinkwrap 파일을 무시하고 package.json을 사용하도록 합니다.

`--ㅜnodedir=/path/to/node/source` Node 소스 코드에 대한 경로를 설정하여 npm이 네이티브 모듈을 컴파일 할 수 있도록 합니다.

`--only={prod[uction]|dev[elopment]}` 인자는 NODE_ENV에 관계없이 devDependencies 만 설치하거나 devDependencies가 아닌 팩키지만 설치하도록 합니다.

알고리즘

npm이 따르는 팩키지설치 알고리즘입니다.

```
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
```

팩키지 의존성이 A{B,C}, B{C}, C{D}라고 하면 이 알고리즘에 따라 다음과 같은 의존성 트리가 만들어 집니다.

```
A
+-- B
+-- C
+-- D
```

A{B,C}, B{C,D@1}, C{D@2}의 경우에는 다음과 같습니다.

```
A
+-- B
+-- C
 `-- D@2
+-- D@1
```

B의 D@1이 최상위에 설치돼어 C는 D@2를 자신의 프라이빗 공간에 설치해야합니다.

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

`npm test` 이후에 `npm install` 을 실행합니다. `npm install` 과 동일한 인자를 입력받습니다.

npm-link

Symlink a package folder

Synopsis

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

alias: npm ln
```

Description

팩키지 링크는 두 단계로 이루어져있습니다.

먼저, 팩키지 폴더 내에서의 `npm link` 는 글로벌로 설치된 `prefix/package-name` 에 대한 심볼릭 링크를 현재 디렉터리에 생성합니다. (`prefix` 에 대해서는 npm-config를 참조)

다음, 다른 위치의 경우에 `npm link package-name` 은 로컬 `node_moduels` 폴더에서 글로벌 실볼릭 링크에 대한 심볼릭 링크를 생성합니다.

디렉터리 이름이 아닌 `package.json` 으로 부터 `package-name` 을 가져옵니다.

팩키지의 이름은 스코프를 포함할 수 있습니다. 이 스코프는 `@` 기호로 시작해야하며 `/` 로 끝나야합니다.

`npm publish` 로 압축파일을 생성할때, 링크된 팩키지에 대해서는 심볼릭 링크로 연결된 현재 상태 그대로 스냅샷이 만들어집니다.

한 번 만들어진 팩키지를 다시 빌드할 필요없이 반복하여 사용할 수 있으므로 자신의 팩키지로 작업할 때 편리할 수 있습니다.

예를 들어

```
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
```

이제 `~/projects/node-redis` 에 어떤 변경이 생기면 `~/projects/node-bloggy/node_modules/node-redis/` 에 반영됩니다. 이 링크는 반드시 디렉터리 이름이 아닌 팩키지 이름이어야만 합니다.

두 단계의 과정을 하번의 과정으로 만들 수 있습니다. 예를 들어 위의 예의 경우 간단히 다음과 같이 할 수 있습니다.

```
cd ~/projects/node-bloggy # go into the dir of your main project
npm link ../node-redis # link the dir of your dependency
```

두 번째 줄의 명령은 다음과 동일합니다.

```
(cd ../node-redis; npm link)
npm link node-redis
```

이 것은 먼전 글로벌 링크를 만들고 글로벌 설치 타겟을 여러분의 프로젝트의 `node_modules` 폴더에 링크합니다.

스코프된 팩키지를 링크하려면, 링크 명령어에 반드시 스코프를 포함시켜야합니다.

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

npm-logout

Log out of the registry

Synopsis

```
npm logout [--registry=<url>] [--scope=<@scope>]
```

Description

토큰 기반 인증을 지원하는 레지스트리에 로그인했을 때 서버에서 해당 토큰의 세션을 만료하도록 요청합니다. 현재 이용중인 환경뿐만 아니라 이 토큰을 사용하는 모든 것들을 무효화시킵니다.

사용자 이름과 패스워드를 기반으로 하는 레거시 레지스트리에 로그인하였다면, 현재 환경상의 암호를 무효화합니다. 이 경우에는 현재 사용중인 환경에만 영향을 줍니다.

`--scope`가 지정되면 해당 스코프에 대한 레지스트리 연결을 찾습니다.

Configuration

registry

Default: https://registry.npmjs.org

npm 팩키지 레지스트리의 기반 URL, `scope`가 지정할 수 있습니다.

scope

Default: none

지정된 스코프에 해당되는 로그인 세션을 지정합니다.

```
npm adduser --registry=http://myregistry.example.com --scope=@myco
```

주어진 스코프에 대한 레지스트리 지정하여 해당 레지스트리에 사용자를 생성하면서 동시에 로그인합니다.

npm-ls

설치된 팩키지의 리스트를 출력

Synopsis

```
npm ls [[<@scope>/]<pkg> ...]

aliases: list, la, ll
```

Description

설치된 모든 버전의 팩키지 목록을 stdout으로 트리구조의 의존성 관계를 표현하여 출력합니다.

인자로 올 수 있는 `name@version-range` 식별자로 주어진 팩키지명에 해당되는 경로에 있는 결과들로만 제한할 수 있습니다. 이 때 내부에 포함된 특정 팩키지의 경로도 같이 보여진다는 점을 기억해야합니다. 예를 들어 `npm ls promzard` 를 npm의 소스 트리내에서 실행하면 다음과 같은 결과가 출력됩니다.

```
npm@@VERSION@ /path/to/npm
└─┬ init-package-json@0.0.4
  └── promzard@0.1.5
```

추가적이거나, 찾을 수 없거나 유효하지 않은 팩키지에 대해서도 출력합니다.

프로젝트에 git URL을 의존 팩키지로 지정하였다면 이 내용은 name@version 다음에 괄호안에 표시되므로 잠재적으로 git fork가 된다는 사실을 알 수 있습니다.

트리 구조는 팩키지간의 의존성을 표현하는 논리적인 의존성 트리를 나타내는 것이며 node_modules 폴더의 물리적인 구조와는 관련이 없습니다.

`ll` 또는 `la` 로 실행하면 추가 정보들을 디폴트로 보여줍니다.

Configuration

json

- Default: false
- Type: Boolean

JSON 형식으로 출력합니다.

long

- Default: false
- Type: Boolean

추가 정보들을 출력합니다.

parseable

- Default: false
- Type: Boolean

트리 구조 대신에 파싱할 수 있는 형태로 출력합니다.

global

- Default: false
- Type: Boolean

현재 프로젝트에 설치된 팩키지 대신에 글로벌로 설치된 팩키지를 출력합니다.

depth

- Type: Int

출력할 의존성 트리의 최대 깊이를 정합니다.

prod / production

- Type: Boolean
- Default: false

`dependencies` 에 지정된 팩키지만 출력합니다.

dev

- Type: Boolean
- Default: false

`devDependencies` 에 지정된 팩키지만 출력합니다.

only

- Type: String

When "dev" or "development", is an alias to **dev**.

When "prod" or "production", is an alias to **production**.`

npm

JavaScript 팩키지 매니저

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

BUGS

When you find issues, please report them:

- web: https:\/\/github.com\/npm\/npm\/issues
- email: 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.

npm-outdated

Check for outdated packages

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.

npm-owner

Manage package owners

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.

npm-pack

Create a tarball from a package

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.

npm-ping

Ping npm registry

Synopsis

npm ping [--registry <registry>]

Description

Ping the configured or given npm registry and verify authentication.

npm-prefix

Display 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.

npm-prune

Remove extraneous packages

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**.

npm-publish

Publish a package

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.

npm-rebuild

Rebuild a package

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.

npm-repo

Open package repository page in the browser

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.

npm-restart

Restart a package

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

npm-root

Display npm root

Synopsis

npm root [-g]

Description

Print the effective **node_modules** folder to standard out.

npm-run-script

Run arbitrary package scripts

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.

npm-search

Search for packages

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.

npm-shrinkwrap

Lock down dependency versions

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.

npm-star

Mark your favorite packages

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.

npm-stars

View packages marked as favorites

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.

npm-start

Start a package

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.

npm-stop

Stop a package

Synopsis

npm stop [-- <args>]

Description

This runs a package's "stop" script, if one was provided.

npm-tag

Tag a published version

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**.

npm-team

Manage organization teams and team memberships

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.

npm-test

Test a package

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 **npat** config to true.

npm-uninstall

Remove a package

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

npm-unpublish

Remove a package from the registry

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**.

npm-update

Update a package

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*.

npm-version

Bump a package 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.

npm-view

View registry info

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, than each printed value will be prefixed with the version it applies to.

If multiple fields are requested, than each of them are prefixed with the field name.

npm-whoami

Display npm username

Synopsis

npm whoami [--registry <registry>]

Description

Print the **username** config to standard output.

npm-folders

npm에서의 폴더 구조

Description

npm은 다양한 것들을 설치하는하는 것이 npm의 임무입니다. 이 장에서는 npm이 어떤 구조로 파일들을 설치하는지 설명합니다.

요약

- 로컬 설치 (디폴트): 현재 팩키지의 최상위 경로 아래 **./node_modules**에 저장합니다.
- 글로벌 설치(-g 옵션을 준 경우) : /usr/local 또는 node가 설치된 경로에 저장됩니다.
- **require()**를 통해서 사용할 것들은 로컬로 설치하십시오.
- 명령행에서 실행할 것들은 글로벌로 설치하십시오.
- 만약 두 가지 경우가 모두 필요하다면 로컬과 글로벌 모두 설치하거나 또는, **npm link**를 사용하십시오.

prefix 설정

prefix 설정에는 node가 설치된 경로가 디폴트 값입니다. 대부분의 시스템에서 이 값은 **/usr/local** 입니다. Unix 시스템에서는 **{prefix}/bin/node** 이지만 윈도우즈 시스템에서는 **{prefix}/node.exe** 입니다.

global 플래그가 지정되면, npm은 이 prefix에 지정된 경로에 설치합니다. **global**이 지정되지 않은 경우에는 현재 팩키지의 최상위 폴더를 사용하며 팩키지가 존재하지 않은 경우에는 현재 작업 디렉터리에 설치합니다.

Node Modules

팩키지는 **prefix** 아래의 **node_modules** 폴더에 들어갑니다. 로컬로 설치하는 경우에는 **require("packagename")**으로 이 팩키지의 메인 모듈을 로드할 수 있게되며, **require("packagename/lib/path/to/sub/module")** 명령으로 그외의 모듈을 로드할 수 있습니다.

글로벌 팩키지들도 같은 방식으로 설치됩니다. 다만 **node_modules** 하위 폴더에 @ 기호로 지정되는 스코프 프리픽스로 그룹이 만들어져 저장되는 점이 다릅니다. 예를 들어 **npm install @myorg/package**는 해당 팩키지를 **{prefix}/node_modules/@myorg/package**위치에 설치합니다.

require()에 필요한 팩키지는 로컬로 설치하십시오.

Executable

글로벌 모드에서는 실행 파일들은 Unix상에서는 **{prefix}/bin**에 링크가 만들어 지며 윈도우즈에는 **{prefix}**에 만들어집니다.

로컬 모드에서는 실행파일들은 **./node_module/.bin**경로에 링크되어 npm의 스크립트에서 실행 가능하게 됩니다. 예를 들어 이 경로에 test 실행 파일이 들어 있으면 **npm test**로 구동할 수 있습니다.

Man Pages

글로벌 모드에서는 man 페이지는 **{prefix}/share/man에 링크됩니다.

로컬 모드에서는 man페이지는 설치되지 않습니다.

윈도우즈에서는 man페이지가 설치되지 않습니다.

Cache

캐시 파일은 Unix 시스템에서 **~/.npm**에, 윈도우즈 시스템에서는 **~/npm-cache**에 저장됩니다.

cache 설정 파라미터로 변경할 수 있습니다.

임시 파일

임시파일은 **tmp**로 설정으로 특정 폴더를 지정할 수 있으며 호나경 변수 TMPDIR,TMP, TEMP에 의해 디폴트가 지정되거나 Unix 상에서 **/tmp**, 윈도우즈에서는 **c:\windows\temp** 입니다.

추가 사항

로컬 설치의 경우, npm은 적절한 **prefix** 폴더를 찾으려고 합니다. 그러므로 **npm install foo@1.2.3** 이라고 하면 다른 폴더로 **cd**한 상태라도 팩키지의 최상위 디렉터리를 찾아서 설치할 것입니다.

$PWD 에서 시작하여, npm은 폴더 트리 구조를 따라 올라가면서 **package.json** 파일이나 **node_modules** 폴더를 포함하고 있는 폴더를 찾습니다. 그렇게 하여 찾은 경로를 npm을 실행한 실질적인 현재 디렉터리로 인식하게 됩니다.

만약 팩키지의 최상위 디렉터리를 찾지 못하면, 현재 디렉터리를 사용합니다.

npm install foo@1.2.3을 실행하면, 팩키지는 캐시에 저장되고 **./node_modules/foo**에 풀립니다. 그리고 foo가 의존하는 팩키지들이 **./node_modules/foo/node_modules/...**에 풀리게 됩니다.

모든 bin 파일들은 **./node_modules/.bin/**에 심볼릭 링크가 되어서 npm 스크립트 내에서 호출할 수 있게됩니다.

글로벌 설치

만약 **global** 설정이 상태라면 npm은 팩키지를 글로벌로 설치합니다.

사이클, 충돌, 폴더 절약

사이클은 **node_modules** 폴더를 찾아가는 과정에서 node의 모듈 시스템의 프로퍼티를 사용하여 처리됩니다. 따라서 각 상태에서 팩키지가 상위 **node_modules** 폴더에 이미 설치되어 있다면, 현재 경로에는 설치되지 않습니다. 예를 들어 **foo -> bar -> baz -> bar -> baz ...**와 같은 팩키지 의존성이 있는 경우라면 **foo/node_modules/bar**에 이미 설치되어 있으므로 **foo/node_modules/bar/node_nmodules/baz/node_modules/** 에는 중복으로 설치하지 않는다는 뜻입니다.

`require("bar")` 의 경우에는 **foo/node_modules/bar**의 위치의 내용을 사용하게 됩니다.

이런 방법은 겹치는 동일 팩키지의 버전이 정확히 일치하는 경우에는 적용됩니다.

Publish

퍼블리쉬 과정에서 npm은 **node_modules** 폴더를 살펴보고 **bundledDependencies**에 지정되지 않은 항목들은 팩키지 압축 파일에 포함되지 않습니다.

이렇게 함으로써 팩키지 개발자가 필요한 의존 팩키지들(개발용 팩키지를 포함하여)을 로컬에 설치할 수 있으며, 다른 곳에서 다운로드 할 수 없는 팩키지만을 선별하여 배포할 수 있도록 합니다.

npmrc

The npm config files

Description

npm은 설정사항들을 명령행, 환경변수, npmrc 파일을 통해 얻습니다.

npm config 명령으로 유저와 글로벌 npmrc 파일을 수정할 수 있습니다.

설정 가능한 모든 옵션들에 대해서는 npm-config를 참조하기 바랍니다.

FILES

네 가지 파일이 관련되어 있습니다.

- 프로젝트별 설정 파일 (\/path\/to\/my\/project\/.npmrc)
- 사용자별 설정 파일 (~\/.npmrc)
- 전역 설정 파일 ($PREFIX\/etc\/npmrc)
- npm 내장 설정 파일 (\/path\/to\/npm\/npmrc)

모든 npm 설정 파일들은 ini 형식으로 **key = value**의 리스트입니다. 환경변수는 **${VARIABLE_NAME}**과 같은 형식으로 참조할 수 있습니다.

```
prefix = ${HOME}/.npm-packages
```

각각의 파일들이 로드되고, 각 설정 옵션들은 우선순위에 따라 결정됩니다. 예를 들어 사용자의 설정 파일에 지정된 값은 전역 설정 파일의 값을 가리게 됩니다.

배열 값은 키 이름에 "[ ]"를 추가하여 지정됩니다.

```
key[] = "first value"
key[] = "second value"
```

주의: 로컬 설정 파일 (프로젝트별, 사용자별 설정파일) **.npmrc**는 중요한 정보를 포함하고 있을 수 있으므로 사용자 게정으로만 읽고 쓰기가 가능하도록 권한 설정이 되어 있어야합니다.

프로젝트별 설정 파일

프로젝트 내부에서 로컬 작업을 할때, 프로젝트의 최상위 경로의 .npmrc 파일(node_modules와 package.json 과 같은 위치에 있는)은 해당 프로젝트에 적용되는 설정 값들을 지정합니다.

이 값들은 npm에서 실행하는 프로젝트의 최상위 경로에만 적용됩니다. 해당 프로젝트가 모듈로 퍼블리쉬되면 효력을 상실합니다. 예를 들어 모듈이 글로벌로만 설치되도록 강제하는 옵션이나 다른 위치에 설치되도록 지정할 수 없다는 뜻입니다.

추가로, `npm install -g` 명령을 실행할 때와 같이 글로벌 모드로 동작할 때에는 이 파일은 읽히지 않습니다.

사용자별 설정 파일

$HOME/.npmrc (또는 userconfig 파라미터가 환경변수나 명령행에서 지정된 경우 그 경로를 따릅니다.)

글로벌 설정 파일

$PREFIX/etc/npmrc (또는 globalconfig 파라미터가 환경변수나 명령행에서 지정된 경우 그 경로를 따릅니다.)

내장 설정 파일

/path/to/npm/itself/npmrc

이 파일은 변경 불가능한 내장 설정 파일입니다.

package.json

Specific of npm's package.json handling

Description

package.json 파일에서 필요한 모든 사항에 대해서 설명합니다. 이 파일은 JSON 파일이어야합니다.

이 글에서 설경하고 있는 많은 사항들이 npm-config에서 설명한 설정 사항들에 의해서 영향을 받습니다.

name

package.json에서 가장 중요한 항목이 name과 version 필드입니다. 필수항목이며 이 값이 없이는 package를 설치할 수 없습니다. name과 version으로 유일하면서 중복되지 않는 식별자를 구성합니다. 팩키지가 변경되면 버전도 같이 변경되어야합니다.

규칙

- 이름은 반드시 214자 이하여야 합니다. 여기에는 스코프 팩키지의 스코프까지 포함됩니다.
- 점이나 밑줄로 시작하면 안됩니다.
- 이름에 대문자가 포함되면 안됩니다.
- 이름은 URL의 일부로 구성되거나 명령행의 인자나 폴더 이름으로 사용되므로 URL에 사용될 수 없는 문자를 포함할 수 없습니다.

팁

- Node의 코어 모듈과 같은 이름을 사용하지 않도록 합니다.
- "js"나 "node"를 이름에 포함시키지 않도록 합니다.
- "require()"를 호출할 때 이름이 파라미터로 전달될 것이므로, 짧지만 어떤 팩키지인지 쉽게 알아 볼 수 있는 이름이 좋습니다.
- npm registry를 확인하여 중복된 이름이 있는지 미리 확인하는 것이 좋습니다.

이름은 스코프로 프리픽를 지정할 수도 있습니다. 예: @myorg/mypackage. 자세한 사항은 npm-scope를 참조하기 바랍니다.

version

package.json에서 가장 중요한 항목이 name과 version 필드입니다. 필수항목이며 이 값이 없이는 package를 설치할 수 없습니다. name과 version으로 유일하면서 중복되지 않는 식별자를 구성합니다. 팩키지가 변경되면 버전도 같이 변경되어야합니다.

버전은 node-semver가 인식할 수 있어야합니다. node-semver는 npm에 함께 포함되어 있습니다. (npm install semver를 통해서 사용할 수 있습니다)

더 자세한 사항은 semver를 참조하기 바랍니다.

description

설명을 추가합니다. 이 설명은 다른 사람들이 npm search로 검색할때 이 팩키지를 찾을 수 있도록 합니다.

keywords

키워드를 추가합니다. 다른 사람들이 npm search로 검색할 수 있도록 합니다.

homepage

프로젝트의 홈페이지 주소를 기록합니다.

주의: 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

프로젝트의 이슈 트랙커의 주소나 이슈를 보고할 수 있는 이메일 주소를 기록합니다. 이 팩키지를 사용하는 사람들이 이슈를 등록할 수 있도록 합니다.

예:

```
{ "url" : "https://github.com/owner/project/issues"
, "email" : "project@hostname.com"
}
```

한가지 항목만 기록하거나 두 가지 값을 모두 쓸 수도 있습니다. 만약 url만을 지정하려면 "bugs" 항목에 객체 대신에 그 값을 직접 문자열로 기록하면 됩니다.

제공된 url은 npm bugs 명령에 사용됩니다.

license

사용자들이 사용상의 허용범위를 확인할 수 있도록 팩키지의 라이센스를 지정해야합니다.

BSD-2-Clause나 MIT와 같은 일반적인 라이센스를 사용한다면 SPDX 라이센스 식별자를 사용하면 됩니다.

```
{ "license" : "BSD-3-Clause" }
```

SPDX 라이센스 아이디의 전체 목록은 https://spdx.org/licenses/에서 확인할 수 있습니다.

복수의 일반 라이센스를 적용하려면 SPDX 문법을 따릅니다. 이 문법에 대해서는 https://www.npmjs.com/package/spdx를 참조합니다.

```
{ "license" : "(ISC OR GPL-3.0)" }
```

SPDX에서 지정되지 않은 라이센스를 사용하거나 수정된 라이센스를 사용하려면 아래와 같은 문자열 값을 지정합니다.

```
{ "license" : "SEE LICENSE IN <filename>" }
```

그리고 `<filename>` 에 지정된 파일을 팩키지의 최상위 폴더에 포함시킵니다.

비공개 팩키지를 다른 사람들이 어떠한 조건에서도 사용하지 못하도록 하려면 다음과 같이 지정합니다.

```
{ "license": "UNLICENSED"}
```

이 경우에는 `"private": true` 를 설정하여 실수로 퍼블리쉬되지 않도록 방지하는 것이 좋습니다.

people fields: author, contributors

"author"는 한 개인을 나타내고 "contributors"는 여러 사람들의 배열을 기록합니다. "person"은 "name" 필드를 가진 객체로써 "url"과 "email"필드는 옵션이며 다음의 예와같습니다.

```
{ "name" : "Barney Rubble"
, "email" : "b@rubble.com"
, "url" : "http://barnyrubble.tumblr.com/"
}
```

또는 이 내용을 모두 단일 문자열로 아래와 같이 기록하면 npm이 각 내용을 파싱할 것입니다.

```
"Barney Rubble <b@rubble.com> (http://barnyrubble.tumblr.com/)"
```

두 가지 방법 모두에서 email과 url은 옵셔널 필드입니다.

files

"files" 필드는 프로젝트에 포함시킬 파일들의 배열입니다. 이 배열에 폴더 이름을 기록하면 (별도의 다른 규칙에 의해 제외하도록 하지 않았다면) 해당 폴더내의 모든 파일들이 포함됩니다.

팩키지 최상위 폴더나 하위 폴더에 ".npmignore" 파일이 있고 여기에 기록된 파일들은 files 배열에 포함되어 있다하더라고 팩키지에서 제외됩니다. .npmignore 파일은 .gitignore파일과 동일한 방법으로 작용합니다.

아래의 파일들은 설정에 관련없이 항상 포함됩니다.

- package.json
- README (및 여기에 해당되는 여러가지 변형파일들)
- CHANGELOG (와 여기에 해당되는 변형 파일들)
- LICENSE / LICENCE

여기에 대비해서 다음의 파일들은 항상 제외됩니다.

- .git
- CVS
- .svn
- .hg
- .lock-wscript
- .wafpickle-N
- *.swp
- .DS_Store
- ._*
- npm-debug.log

main

main 필드는 프로그램의 시작 포인트를 가리키는 모듈 ID입니다. 팩키지의 이름이 **foo** 라고 한다면 이 팩키지를 설치한 사용자가 **require("foo")**를 호출하면 팩키지의 메인 모듈이 익스포트(*export*)한 객체를 반환받게 됩니다.

모듈 ID는 팩키지 폴더의 최상위를 기준으로한 상대 경로이어야합니다.

bin

많은 팩키지들이 PATH에 추가되어야하는 하나 이상의 실행 파일을 포함하고 있습니다. npm은 이런 작업을 손쉽게 합니다. (실제로 npm 실행 파일 자체도 이 기능을 사용합니다.)

이 기능을 사용하려면 package.json내에 bin 필드를 작성하여 명령어 이름과 로컬 파일 이름과의 맵핑을 해야합니다. 설치과정에서 npm은 글로벌 설치일 경우 prefix/bin 내에, 로컬 설치의 경우 ./node_modules/.bin 폴더 내에 심볼릭 링크 파일을 생성합니다.

예를 들어 myapp에 다음 처럼 bin 필드를 작성하면

```
{ "bin" : { "myapp" : "./cli.js" } }
```

myapp을 설치하면 npm은 cli.js 스크립트를 /usr/local/bin/myapp으로 심볼릭 링크를 생성합니다.

하나의 실행파일만 있고, 이 실행 파일의 이름이 팩키지 이름과 동일하다면 간단히 문자열로 지정할 수도 있습니다.

```
{ "name": "my-program"
, "version": "1.2.5"
, "bin": "./path/to/program" }
```

이런 경우 실제로 다음과 동일한 결과입니다.

```
{ "name": "my-program"
, "version": "1.2.5"
, "bin" : { "my-program" : "./path/to/program" } }
```

man

man 프로그램이 찾을 수 있는 위치에 설치할 하나의 파일 이름이나 파일이름의 배열을 지정합니다.

하나의 파일만 지정하고 이 파일이 **man** 의 결과이어야 한다면 이 파일의 실제 파일명은 어떤 이름으든 상관이 없습니다. 예를 들어

```
{ "name" : "foo"
, "version" : "1.2.3"
, "description" : "A packaged foo fooer for fooing foos"
, "main" : "foo.js"
, "man" : "./man/doc.1"
}
```

이 경우에 `./man/doc.1` 파일은 `man foo` 의 대상이 됩니다.

만약 파일명이 아래와 같이 팩키지명으로 시작되지 않으면 팩키지명이 자동으로 접두어(*prefix*)로 붙게 됩니다.

```
{ "name" : "foo"
, "version" : "1.2.3"
, "description" : "A packaged foo fooer for fooing foos"
, "main" : "foo.js"
, "man" : [ "./man/foo.1", "./man/bar.1" ]
}
```

이 경우 **man foo**와 **man foo-bar**에 대응하는 파일이 생성됩니다.

man 파일명은 숫자로 끝나야만 하며 옵션사항으로 압축된 파일의 경우 **.gz** 접미사(*suffix*)가 가능합니다. 이때 숫자는 이 파일이 설치되는 man 섹션을 뜻합니다.

```
{ "name" : "foo"
, "version" : "1.2.3"
, "description" : "A packaged foo fooer for fooing foos"
, "main" : "foo.js"
, "man" : [ "./man/foo.1", "./man/foo.2" ]
}
```

이 경우는 **man foo**와 **man 2 foo** 엔트리를 생성합니다.

directories

CommonJS 팩키지 스펙에 규정된 팩키지에 적용할 수 있는 다수의 **directories** 객체의 구조가 설명되어 있습니다.

directories.lib

사용자들에게 라이브러리가 존재하는 장소를 알려줍니다. 특별한 작업을 수행하지 않으며 단순히 메타 정보를 나타냅니다.

directories.bin

directories.bin에 bin 디렉터리를 지정하면 해당 폴더에 들어 있는 모든 파일들이 추가됩니다.

bin 지시자가 동작하는 방식 때문에 **bin**과 **dictories.bin**이 동시에 사용하는 것은 오류입니다. 개별 파일들을 각 각 지정하려면 **bin**을 사용하고 이미 존재하는 bin 디렉터리의 모든 파일을 추가하려면 **directories.bin**을 사용하도록 하십시오.

directories.man

전체 man 페이지를 담고 있는 폴더입니다.

directories.doc

마크다운(*markdown*) 파일을 저장하는 폴더입니다.

directories.example

예제 스크립트를 저장하는 곳입니다.

repository

코드를 저장하는 저장소를 지정합니다. 도움을 제공하고자하는 사람들에게 소스 저장소 위치를 알려주게 됩니다. 만약 이 레포지토리가 GitHub라면 **npm docs** 명령어가 이 저장소의 소유자를 찾을 수 있습니다.

아래와 같이 합니다.

```
"repository" :
 { "type" : "git"
 , "url" : "https://github.com/npm/npm.git"
 }

"repository" :
 { "type" : "svn"
 , "url" : "https://v8.googlecode.com/svn/trunk/"
 }
```

이 URL은 (읽기 전용이라도) 공개되어 있어서(*pulbic*) VCS 프로그램에서 별도의 수정없이 접근 가능해야합니다. 브라우저에 입력하는 HTML로된 프로젝트 페이지 URL이 아닙니다.

GitHub, GitHub gist, Bitbucket, GitLab 레포지토리의 경우에는 **npm install**에 사용되는 단축 문법과 동일한 방식을 사용할 수 있습니다.

```
"repository": "npm/npm"

"repository": "gist:11081aaa281"

"repository": "bitbucket:example/repo"

"repository": "gitlab:another/repo"
```

scripts

"scripts" 프로퍼티는 딕셔너리 형태로 팩키지의 라이프 사이클상에서 다양한 순간에 실행되는 스크립트 명령어들을 담고 있습니다. 이때 키는 라이프사이클 이벤트이고, 값은 이 때 실행할 명령어입니다.

config

"config" 객체를 통해서 팩키지 스크립트에서 사용되는 설정 파라미터들을 지정할 수 있습니다. 다음의 예를 들면...

```
{ "name" : "foo"
, "config" : { "port" : "8080" } }
```

"start" 명령에서 **npm_package_config_port**를 참조한다고 가정하면, 이 값은 **npm config set foo:port 8001** 명령으로 오버라이드할 수 있습니다.

dependencies

팩키지 이름과 버전 범위를 맵핑한 오브젝트를 통해서 의존성을 지정합니다. 버전 범위는 하나 또는 공백으로 구분되는 문자열들로 나타냅니다. 의존성은 압축파일 형태이거나 git URL로 식별됩니다.

여기에 시험목적이거나 코드 변환이나 컴파일을 위한 도구는 dependencies 객체에 포함시키지 말도록 하고, 대신에 devDependencies를 참조하기 바랍니다.

버전 범위를 지정하는 것이 자세한 사항들은 semver를 참조하기 바랍니다.

- `version` : version에 정확히 일치해야합니다.
- `>version` : 지정된 version보다 더 커야합니다.
- `>=version`
- `<version`
- `<=version`
- `~version` : 해당 버전에 대략적으로 일치해야합니다.
- `^version` : 해당 버전과 호환이되어야 합니다.
- `1.2.x` : 1.2.0, 1.2.1 등등, 1.3.0은 안됩니다.
- `http://...` : URL 의존성 참고
- `*` : 모든 버전와 매칭됩니다.
- `""` : (빈 문자열) *과 동일합니다.
- `version1 - version2` : `>=version1 <=version2` 와 동일합니다.
- `range1 || range2` : range1 또는 range2중에 하나를 만족하면 통과합니다.
- `git...` : Git URL 의존성 참고
- `tag` : 태그되었거나 tag로 퍼블리쉬된 버전을 지정합니다.
- `path/path/path` : 로컬 경로를 참고하기 바랍니다.

아래의 예는 모두 유효한 dependencies 정의입니다.

```
{ "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"
 }
}
```

URL 의존성

버전 범위 대신에 압축 파일의 URL을 지정할 수 있습니다. 이 압축파일은 팩키지가 인스톨될 때 다운로드되어 함께 로컬에 설치될 것입니다.

Git URL 의존성

Git URL은 다음과 같은 형태입니다.

```
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
```

여기서 **commit-ish** 부분은 `git checkout` 의 인자가 될 수 있는 어떤 태그나 브랜치가 될 수도 있습니다. 디폴트는 **master**입니다.

GitHub URL

버전 1.1.65부터 GitHub URL을 `"foo":"user/foo-project"와 같은 간단한 방법으로 참조할 수 있게 되었습니다. Git URL에서의 **commit-ish**와 같은 방법이 그대로 적용됩니다.

```
{
 "name": "foo",
 "version": "0.0.0",
 "dependencies": {
 "express": "visionmedia/express",
 "mocha": "visionmedia/mocha#4727d357ea"
 }
}
```

로컬 경로

버전 2.0.0에서 부터 팩키지를 저장하고 있는 로컬 디렉터리 경로를 사용할 수 있습니다. 로컬 경로는 **npm install -S** 또는 **npm install --save** 명령을 통해서 저장될 수 있습니다. 아래와 같은 형태로 사용가능합니다.

```
../foo/bar
~/foo/bar
./foo/bar
/foo/bar
```

어떤 방법이든 경로는 상대경로로 노말라이즈 되어 **package.json**에 추가됩니다.

```
{
 "name": "baz",
 "dependencies": {
 "bar": "file:../foo/bar"
 }
}
```

이 기능은 공개 레지스트리에 퍼블리쉬하면 안되는 팩키지를 사용해서 로컬에서 오프라인상태로 개발이 가능하도록 합니다.

devDependencies

어떤 사람이 우리의 모듈을 이용하여 다른 프로그램을 작성할 때 우리 모듈에서 사용하고 있는 외부 테스트 프레임워크나 문서화 프레임워크를 같이 다운로드 받고 싶지 않을 것입니다. 이러한 프레임워크는 우리 모듈을 개발할 때만 필요한 것이지 우리 모듈을 사용할 때에는 필요하지 않기 때문입니다.

이런 경우, 이런 추가적인 항목들을 **devDependencies** 객체에 넣는게 최선입니다.

이러한 추가 항목들은 팩키지의 최상위 경로에서 **npm link** 또는 **npm install**을 실행할 때 설치되며, 다른 npm 설정 파라미터와 동일한 방법으로 다뤄집니다.

CoffeeScript오 같은 JavaScript로 컴파일되는 플랫폼 독립적인 빌드 과정들은 **prepublish** 스크립트를 이용하도록 하며 필요한 팩키지들은 devDependency에 추가합니다.

예:

```
{ "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"
}
```

prepublish 스크립트는 퍼블리쉬되기 전에 실행되므로 퍼블리쉬하기 전에 매번 직접 컴파일하지 않아도 됩니다. 개발 모드 (**npm install**을 로컬에서 실행하는)에서도, 이 스크립트가 실행되므로 쉽게 테스트할 수 있습니다.

peerDependencies

bundledDependencies

optionalDependencies

engines

호환되는 node의 버전을 지정할 수 있습니다.

```
{ "engines" : { "node" : ">=0.10.3 <0.12" } }
```

dependencies처럼 버전을 지정하지 않거나 (또는 `*` 을 지정하면) 모든 버전의 node를 뜻합니다.

"engines" 필드를 지정하면 npm은 "node"가 리스트에 반드시 포함되어야합니다. "engine"이 생략되면 npm은 디폴트로 node를 사용하는 것으로 가정합니다.

"engines"필드를 통해서 우리의 프로그램이 정상적으로 동작할 수 있는 npm의 버전을 지정할 수도 있습니다.

```
{ "engines" : { "npm" : "~1.0.20" } }
```

사용자가 **engine_strict** 설정 플래그를 지정하지 않는한 이 필드는 참고사항으로만 사용됩니다.

engineStrict

이 기능은 npm 3.0.0에서 디프리케이트 되었습니다.

npm 3.0.0 이전에는 사용자가 **engine-strict** 플래그를 지정한 것과 동일하게 간주되었습니다.

os

우리의 모듈이 동작할 수 있는 운영 체제를 지정할 수 있습니다.

```
"os" : [ "darwin", "linux" ]
```

운영체제를 화이트리스트 방식 대신에 `!` 문자를 사용해서 블랙리스트 방식으로도 지정할 수 있습니다.

```
"os" : [ "!win32" ]
```

호스트의 운영 체제는 **process.platform**에 의해서 결정됩니다.

cpu

동작 가능한 cpu 아키텍쳐를 지정합니다.

```
"cpu" : [ "x64", "ia32" ]
```

os 옵션과 마찬가지로 블랙리스트 방식이 가능합니다.

```
"cpu" : [ "!arm", "!mips" ]
```

호스트의 아키텍쳐는 **process.arch**로 결정됩니다.

preferGlobal

작성하는 팩키지가 명령행 애플리케이션으로 반드시 글로벌로 설치되어야만 한다면, 이 값을 **true**로 설정해서 로컬로 설치되는 경우에 경고를 출력할 수 있습니다.

이 것을 설정한다고 사용자가 로컬로 설치하는 것을 금지할 수는 없지만 혼란이 발생하는 것을 방지할 수는 있습니다.

private

package.json에서 이 값을 `"private": true`로 설정하면 npm이 이 팩키지를 퍼블쉬할 수 없게됩니다.

프라이빗 레포지토리를 실수로 공개하는 것을 방지할 수 있습니다. 만약 이 팩키지가 특정 레지스트리 (예를 들어 회사 내부의 레지스트리)에만 퍼블리쉬되어야한다면 **publishConfig** 딕셔너리의 **registry** 설정 파라미터를 사용해야합니다.

publishConfig

퍼블리쉬 과정에서 사용되는 설정 값들입니다. 태그나 레지스트리와 접근 권한을 지정할 때 사용하여 팩키지가 디폴트로 "latest"로 태그 되거나, 글로벌 공개 레지스트리로 퍼블리쉬되거나, 스코프 지정된 모듈이 프라이빗이 되는 것을 방지할 수 있습니다.

어떠한 config 값들도 오버라이드할 수 있으나, "tag", "registry", "access" 가 퍼블리쉬 과정에서 가장 중요한 사항일 것입니다.

디폴트 값들

npm은 팩키지의 내용물에 따라 디폴트 값을 사용합니다.

- `"scripts": {"start": "node server.js"}`

  server.js 파일이 팩키지의 최상위 경로에 존재하면 npm은 start 명령으로 `node server.js` 를 디폴트로 사용합니다.

- `"scripts":{"preinstall": "node-gyp rebuild"}`

  binding.gyp 파일이 팩키지 최상위 경로에 존재하면 npm은 preinstall 명령에 node-gyp을 사용하여 컴파일합니다.

- `"contributors": [...]`

  AUTHORS 파일이 팩키지 최상위 경로에 존재하면 npm은 파일의 각 라인을 **Name (url)** 형태로 간주합니다. #으로 시작하는 라인이나 공백라인은 무시합니다.