1. npm 명령어의 종류

NPM 이란?

npm 은 자바스크립트 프로그래밍 언어를 위한 패키지 관리자이다. 자바스크립트 런타임 환경 Node.js의 기본 패키지 관리자이다

NPM 명령어 종류

npm init



## create a package.json file

## package.json 파일을 만듭니다

npm의 첫 시작인 초기화 작업을 하는 명령어 입니다.

**package.json**은 프로젝트 정보와 의존성(dependencies)을 관리하는 문서입니다.

이미 작성된 **package.json** 문서는 어느 곳에서도 동일한 개발 환경을 구축할 수 있게 해줍니다.

JSON 포맷으로 작성해야 하며, 다음과 같은 옵션들이 추가될 수 있습니다.

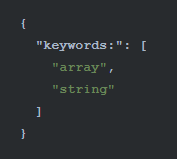
**name**

URL이나 Command Line의 일부로 사용될 소문자로 표기된 214자 이내의 프로젝트(패키지) 이름으로, 간결하고 직관적인 이름으로 설정하되 다른 모듈과 동일한 이름을 피하세요.

**version**[SemVer(The semantic versioner for npm)](https://docs.npmjs.com/misc/semver)로 분석 가능한 형태의 버전을 지정합니다.

**description**프로젝트(패키지)의 설명을 지정합니다  
(npm search 사용 시 도움이 됩니다.)

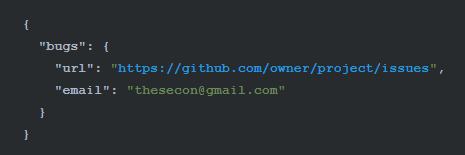
### keywords 프로젝트(패키지)의 키워드를 배열로 지정합니다. (npm search 사용 시 도움이 됩니다.)



### homepage 프로젝트 홈페이지로 연결되는 URL을 지정합니다.

### bugs

패키지에 문제가 있을 때 보고될 이슈 트래커(추적시스템) 및 이메일 주소 등에 대한 URL을 지정합니다.



### license 패키지 사용을 허용하는 방법과 제한 사항을 알 수 있도록 라이센스를 지정합니다.

[Open Source Licenses](https://opensource.org/licenses)

### author

제작자의 이름을 지정합니다.

### files

패키지가 의존성으로 설치될 때 같이 포함될 파일들의 배열입니다.  
생략하면 자동 제외로 설정된 파일을 제외한 모든 파일이 포함됩니다



### main 프로그램의 기본 진입 점(entry point)를 지정합니다. 패키지의 이름이 jquery이고, 사용자가 require('jquery')를 사용하면 진입 점의 메인 모듈에서 exports object가 반환(return)됩니다.

### repository

코드가 존재하는 장소를 지정합니다.

GitHub를 사용하면 **npm docs** 명령을 사용하여 찾을 수 있습니다.



### script

패키지 라이프 사이클에서 여러 번 실행되는 스크립트 명령을 포함합니다.



### dependencies

패키지의 배포 시 포함될 의존성 모듈을 지정합니다.

### devDependencies

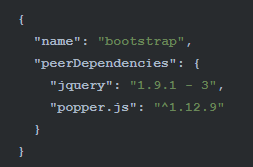
패키지의 개발 시 사용될 의존성 모듈을 지정합니다.

(배포 시 포함되지 않습니다)

### peerDependencies

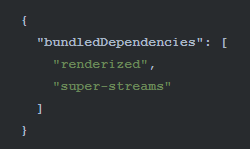
패키지의 호환성 모듈을 지정합니다.

(npm@3 이후로 배포 시 포함되지 않습니다, 대신 호환성 모듈이 없으면 경고 메시지가 표시됩니다)



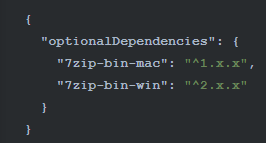
### bundledDependencies 패키지를 게시할 때 번들로 묶을 패키지 이름을 배열로 지정합니다.

npm 패키지를 로컬에서 보존해야 하거나 단일 파일 다운로드를 통해 사용할 수있는 경우 **npm pack**을 실행하여 패키지를 **<name>-<version>.tgz** 형태의 [tarball 파일](https://ko.wikipedia.org/wiki/Tar_(%ED%8C%8C%EC%9D%BC_%ED%8F%AC%EB%A7%B7))로 묶을 수 있습니다.



### optionalDependencies

npm을 찾을 수 없거나 설치에 실패한 경우 계속 진행하려면 optionDependencies 객체에 넣을 수 있습니다.



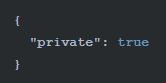
dependencies 동일하게 배포 시 포함될 의존성 모듈을 지정하지만, 빌드 실패로 인해 설치 과정이 중단되지 않습니다.

### engines

패키지가 작동하는 Node 버전을 지정합니다.

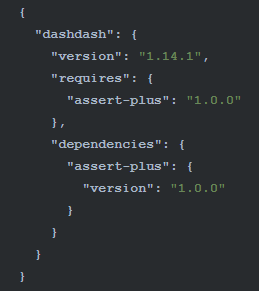
### private

개인 저장소의 우연한 발행을 방지하기 위해 npm에서 패키지의 공개 여부를 지정합니다.



## **package-lock.json**

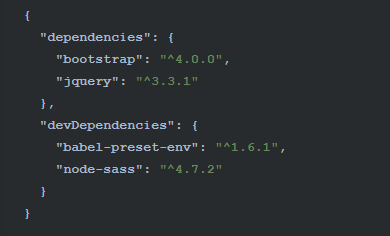
**package.json**이 동일한 개발 환경 구축을 위한 정보를 가지고 있지만, 다양한 경우에 의해 동일한 개발 환경 구축에 문제가 발생할 수 있습니다.

예를 들어, 다음과 같은 어떤 상위 모듈(**dashdash**)에서 사용하는 하위 모듈 중 하나(**assert-plus**)의 버전이 변경되었다고 가정합니다.  
그렇다면 상위 모듈 버전이 동일하다고 하더라도 내부적으로 다른 결과를 출력할 수 있습니다.  


이를 방지하기 위해 npm으로 **node\_modules**의 구성 트리 또는 **package.json**을 수정하는 모든 작업에 대해 **package-lock.json**이 자동으로 생성됩니다.

모든 작업에 대해 자동 생성하므로 의존성 업데이트와 같은 버전 변경에 대해서도 동일한 모듈 트리를 생성할 수 있게 됩니다.

# **install (패키지 설치)**



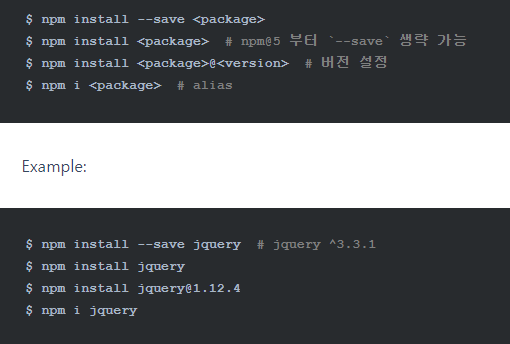
## **이미 정의된 의존성 모듈 설치**

**package.json** 파일에 이미 정의된 의존성(**dependencies**) 모듈(Modules)이 있으면, 나열된 모든 모듈을 로컬 **node\_modules** 폴더에 설치합니다.



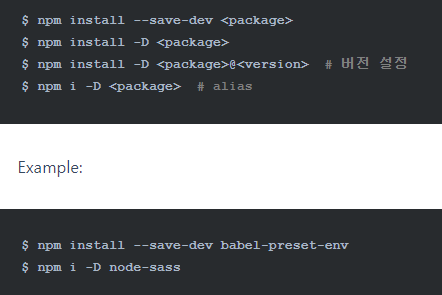
**아직 정의되지 않은 의존성 모듈 설치**

### --save

프로젝트(패키지)가 배포(Deploy) 시 사용될 의존성 모듈을 정의하고 설치합니다.  
**--save** 플래그(Flag)를 사용하거나 생략합니다.  
**"dependencies"**에 나열됩니다.  


### --save-dev / -D

개발 단계에서만 사용하는 의존성 모듈을 정의하고 설치합니다.

**-D**(**--save-dev**) 플래그를 사용합니다. **"devDependencies"**에 나열됩니다.  


### --save-optional / -O

선택적 의존성 모듈을 정의하고 설치합니다.

**-O**(**--save-optional**) 플래그를 사용합니다.

**"optionalDependencies"**에 나열됩니다.

### --save-exact / -E

npm의 기본 SemVer 연산자(**^**, **~** 같은)를 사용하는 대신 정확한 버전으로 설치합니다.  
**-E**(**--save-exact**) 플래그를 사용합니다.  
**"dependencies"**에 나열됩니다.  


### --save-bundle / -B

번들로 묶을 패키지 의존성 모듈을 정의하고 설치합니다.

**-B**(**--save-bundle**) 플래그를 사용합니다.

**"bundledDependencies"**에 나열됩니다.

### 전역 설치

패키지를 command line tool로 사용하려면 전역(Global)으로 설치할 수 있습니다.

전역으로 설치된 패키지는 디렉토리에 관계없이 작동합니다.

**-g** 플래그를 사용합니다.



# **기타 자주 사용하는 CLI commands**

## **outdated (오래된 패키지 확인)**

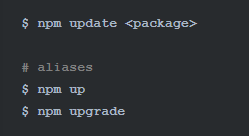
설치된 패키지가 현재 구형(오래된 패키지)인지 여부를 확인하기 위해 레지스트리를 검사합니다.



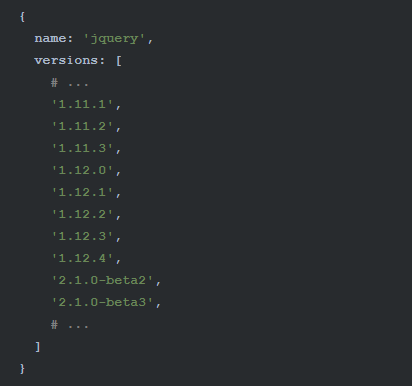
## **update (버전 업데이트)**

패키지를 Semver에 맞는 최신 버전으로 업데이트합니다.

패키지 이름을 지정하지 않으면 지정된 위치(Global/Local)의 모든 패키지를 업데이트합니다.



예를 들어, 다음과 같은 버전 정보를 가지는 **jquery** 모듈을 업데이트할 경우를 살펴봅시다.



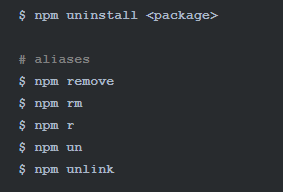
다음과 같이 모듈 버전에 범위 연산자를 **^**(Caret)으로 사용했다면, Minor Level 범위에서 가장 최신 버전으로 업데이트합니다.

따라서 **1.12.4** 버전을 설치합니다.

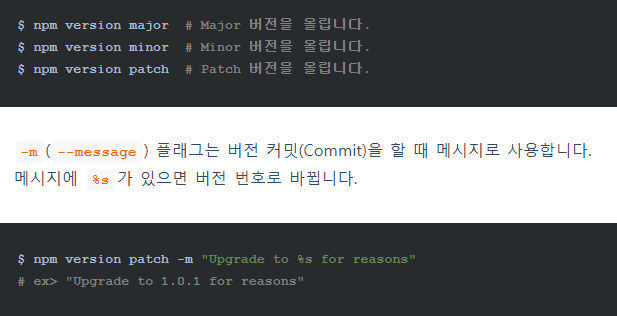
## **uninstall (패키지 제거)**

설치된 패키지를 제거합니다.

**npm install**과 동일하게 **--save**, **--save-dev** 같은 플래그들을 사용할 수 있습니다.

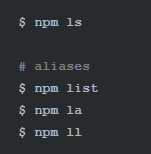


## **version (패키지 버전 관리)**

자신의 프로젝트(패키지)의 버전을 관리하기 위해서 **npm version**을 사용합니다.  


## **ls (패키지 목록)**

설치된 패키지를 나열합니다.



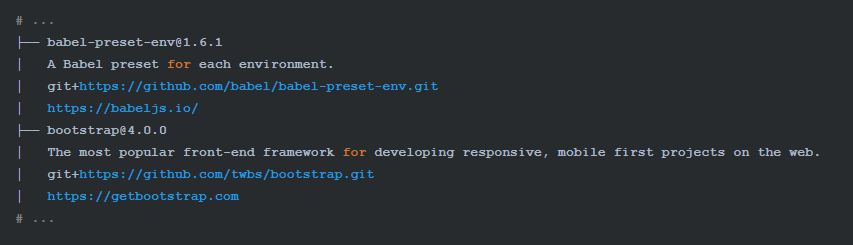
### --json

패키지 정보를 JSON 형식으로 표시합니다.



### --long

패키지의 확장 정보를 표시합니다.



### --depth

의존성 트리를 표시하는 최대 깊이를 정수(Integer)로 지정합니다.



### --prod / --production

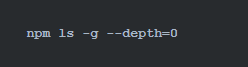
**dependencies** 의존성 모듈만 표시합니다.

### --dev

**devDependencies** 의존성 모듈만 표시합니다.

### 전역 설치된 패키지 목록

전역 설치된 패키지들의 목록을 나열합니다.

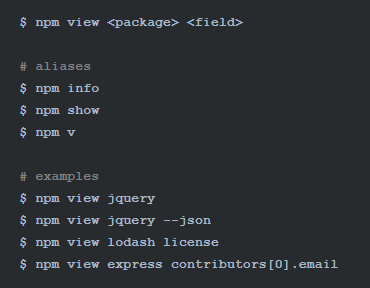


## **view (패키지 정보 보기)**

패키지에 관한 데이터(정보)를 보여줍니다.

필드가 객체이면 JavaScript 객체 리터럴로 출력됩니다.

**--json** 플래그가 주어지면, JSON 형식으로 출력됩니다.



## 

npm 공식 기술문서 ( 모든 명령어 존재 )

file:///C:/Program%20Files/nodejs/node\_modules/npm/docs/public/cli-commands/npm/index.html

<https://heropy.blog/2018/02/18/node-js-npm/>

2. npm 옵션 중 -g, --save 의 의미는 무엇인가.

1. -g: 패키지를 전역에 설치하는 명령어

$ npm install node-emoji;

$ npm install -g node-emoji;

* 옵션을 별도로 지정하지 않을 경우 기본적으로 지역 설치되며 이렇게 설치된 패키지는 해당 프로젝트 안에서만 사용할 수 있음. → 루트 디렉토리에 node\_modules 폴더 생성하고 그 안에 패키지 설치
* 패키지을 전역에 설치할 경우 전역에서 참조할 수 있으며 여러 프로젝트가 공통으로 사용하는 패키지 설치할 때 이용
* 전역에 설치된 패키지는 OS에 따라 설치 장소가 다름
  + macOS: /usr/local/lib/node\_modules
  + 윈도우: c:\Users\%USERNAME%\AppData\Roaming\npm\node\_modules

1. --save

* Node.js 프로젝트에서는 많은 패키지를 사용하고 패키지 버전도 빈번하게 업데이트 되므로 프로젝트가 의존하고 있는 패키지를 일괄 관리해야 함
* npm은 package.json 파일을 통해서 프로젝트 정보와 패키지 의존성(dependency) 관리 → 미리 작성된 package.json 파일을 배포하면 동일한 작업 환경 구축할 수 있음
* package.json 생성하는 방법
  + 프로젝트 루트에서 npm init 실행 → 프로젝트에 관한 정보 입력
  + 기본 설정값으로 생성된 package.json 파일을 수정하는 방법이 더 편리할 수 있으므로 npm init 명령어에 --yes 또는 -y 옵션을 추가

$ npm init -y

Wrote to /Users/leeungmo/Desktop/emoji/package.json:

{

"name": "emoji",

"version": "1.0.0",

"description": "",

"main": "index.js",

"dependencies": {

"node-emoji": "^1.10.0"

},

"devDependencies": {},

"scripts": {

"test": "echo \"Error: no test specified\" && exit 1"

},

"keywords": [],

"author": "",

"license": "ISC"

}

* 가장 중요한 내용은 name과 version → 고유성 판단 기준, 생략 불가
* dependencies: 해당 프로젝트가 의존하는 패키지들의 이름과 버전을 명시한다. 여기서 의존하는 패키지란 해당 프로젝트에서 참조하는 모듈을 의미한다. 프로젝트를 진행할 때는 이미 만들어진 여러 패키지를 참조해서 사용하는데, package.json 파일의 dependencies 항목에 해당 패키지의 이름과 버전을 명시함으로써 의존성을 설정
* npm install 명령어에 --save 옵션을 사용하면 패키지 설치와 함께 package.json의 dependencies에 설치된 패키지 이름과 버전이 기록됨

$npm install --save node-emoji;

* npm@5부터 --save는 기본 옵션이 되었다. --save 옵션을 사용하지 않더라도 모든 install 명령은 package.json의 dependencies에 설치된 패키지와 버전을 기록한다. 기존의 --save-dev은 변경되지 않았다.

**3. 리액트 클래스구조에서 export, import의 의미는 무엇인가.**

파일이나 모듈안의 함수나, 객체를 export 하는데 사용됩니다. ES6에서 정의된 API로, 브라우저에서 기본으로 제공되지 않기 때문에, Traceurur Complier, Babel, Rollup, Webpack과 같은 별로의 트랜스파일러와 함께 사용되어야 합니다. export로 함수, 객체, 값을 내보내고 import로 가져다 씁니다. 내보내는 모듈은 "use strict"의 존재 유무와 상관없이 무조건 엄격 모드입니다. export 문은 HTML 안에 작성한 스크립트에서는 사용할 수 없습니다.

**모듈(Module)**모듈이란 여러 기능을 담고 있는 코드를 담고 있는 여러 파일을 import, export 명령어를 사용하여 해당 파일 안에서 특정 코드만 가져와서 작업하는 방식을 의미합니다.

**엄격모드(StrictMode**)  
strict 모드는 ES5(ECMA Script 5)에 추가된 키워드입니다.strict 모드는 자바스크립트가 묵인했던 에러들의 에러 메시지를 발생시킵니다. 엄격하게 문법 검사를 하겠다.. 로 이해하면 될 것 같습니다.

**Export**내보내기에는 두 종류, 유명(named)과 기본(default) 내보내기가 있습니다.

**유명 내보내기 (export — named)**

*// 먼저 선언한 식별자 내보내기*

export { myFunction, myVariable };

*// 각각의 식별자 내보내기*

*// (변수, 상수, 함수, 클래스)*

export let myVariable = Math.sqrt(2);

export function myFunction() { ... };

*//내보낼때 as로 이름변경하기*

export { myFunction as function1,

         myVariable as variable };

유명 내보내기는 여러 값을 내보낼 때 유용합니다.  
named export는 동일한 이름으로 가져올수 있고, 이름을 바꾸려면 export할때 as를 사용해야합니다.

**기본 내보내기 (export — default)**

*// 먼저 선언한 식별자 내보내기*

export { myFunction as default };

*// 각각의 식별자 내보내기*

export default function () { ... };

export default class { ... }

*// test.js*

let k; export default k = 12;

*// 임의의 다른 파일*

import m from './test'; *// k가 기본 내보내기이므로, 가져오는 이름으로 k 대신 m을 사용해도 문제 없음*

console.log(m);         *// 12 기록*

하나의 모듈(파일)에서는 하나의 default export만 가능합니다.  
기본 내보내기는 어떤 이름으로도 가져올 수 있습니다.

**Import**외부 스크립트 또는 외부 모듈의 export된 함수, 객체를 가져오는데 사용됩니다. export와 마찬가지로, ES6에서 정의된 API 입니다. import는 모든 브라우저에서 기본으로 제공되지 있지 않기 때문에, Traceur Compiler, Babel, Rollup, Webpack과 같은 별도의 트랜스파일러와 함께 사용되어야 합니다.

import name from "module-name";

import \* as name from "module-name";

*//name : 가져온 값을 받을 객체 이름.*

import { member } from "module-name";

import { member as alias } from "module-name";

import { member1, member2 } from "module-name";

import { member1, member2 as alias2, [...] } from "module-name";

*//member, memberN : export 된 모듈에서 멤버의 이름*

import defaultMember, { member [, [...]] } from "module-name";

import defaultMember, \* as alias from "module-name";

import defaultMember from "module-name";

*//defaultMember : 기본내기로 export 된 모듈의 default 이름*

*//alias, aliasN : export된 멤버의 이름을 지정한 이름*

import "module-name";

*//module-name : 가져올 모듈 이름 (파일명)*