



# NPM

## What?

- A package manager for Node
- Acts as a command line tool to add, remove , update packages for the Node application.
- Only works with Node, can't work with Python, Java etc.
- Backed by Microsoft.

## Yarn vs NPM

- Another package manager like NPM
- Backed by Facebook(Meta)
- Was created to improve upon NPM. It is a bit faster.
- Some packages assume that NPM is installed on the machine, and might break if it is not.

## Installing NPM

- NPM uses JS internally, and requires Node to function. Hence download and install Node.js
- On Windows, NPM is installed when you install Node.

```
>node -v //gives the version of the node installed  
>npm -v //gives the version of the NPM installed
```

## Installing Packages

- When we write

```
>npm install lodash{//here lodash is the name of a package}
```

- NPM goes to its server, downloads its compressed code files and extracts it into **node\_modules** folder.

## How NPM Installs Packages

- If we run

```
>npm view lodash
```

- It gives out the following output

```
C:\Users\rajee\OneDrive\Desktop\npm_basics>npm view lodash
lodash@4.17.21 | MIT | deps: none | versions: 114
lodash modular utilities.
https://lodash.com/

keywords: modules, stdlib, util

dist
.tarball: https://registry.npmjs.org/lodash/-/lodash-4.17.21.tgz
.shasum: 679591c564c3bffaee8454cf0b3df370c3d6911c
.integrity: sha512-v2kDEe571ecTulaDIuNTPy3Ry4gLGJ6Z103vE1krqXZNrsQ+LFTGHVxVjcXPs17LhbZVGedAJv8XZ1tvj5FvSg==
.unpackedSize: 1.4 MB

maintainers:
- mathias <mathias@qiwi.be>
- jdalton <john.david.dalton@gmail.com>
- bnjmnt4n <benjamin@dev.ofcr.se>

dist-tags:
latest: 4.17.21

published a year ago by bnjmnt4n <benjamin@dev.ofcr.se>
C:\Users\rajee\OneDrive\Desktop\npm_basics>
```

- Notice the .tarball property, it is the URL of the .tar file(compressed package file) of the package.
- When a developer publishes a package on NPM, he uploads a .tar file of the package, and NPM only stores that on its server.
- So when we write

```
>npm install lodash
```

- NPM goes to the .tarball URL and downloads the file, and then unzips it into the **node\_modules** folder.

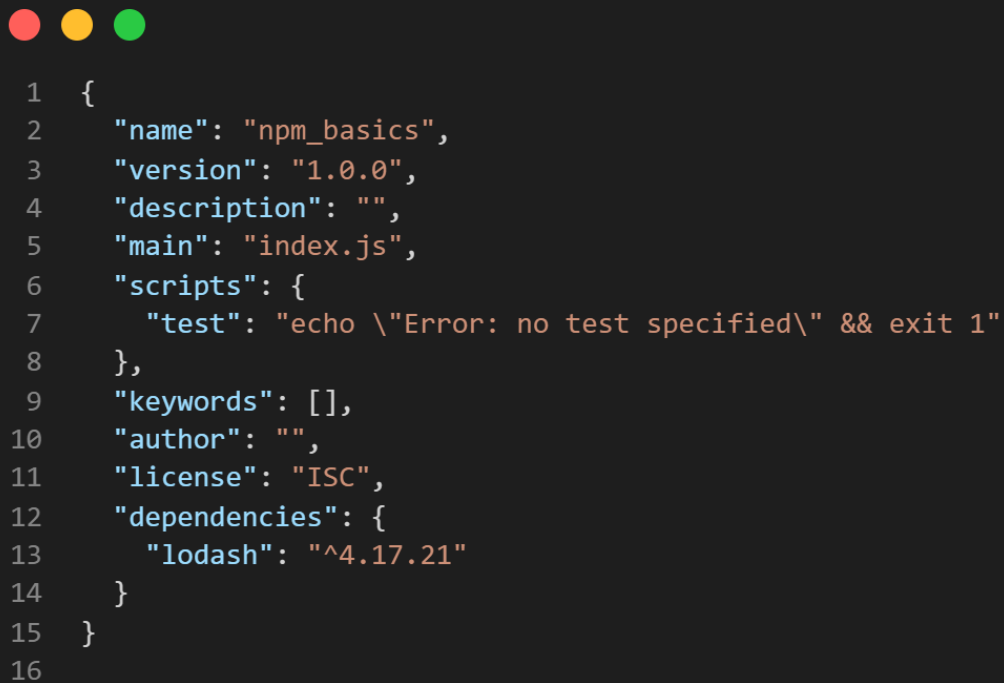
- NPM also checks the integrity of the downloaded package.

## Package.json

- NPM keeps a special file named package.json to keep track of the packages that are currently present in the node\_modules folder.
- Create this file using

```
>npm init -y  
  
//here -y means yes to all the questions asked
```

- Or it gets created when we install any package. But it is good to use the above command since it adds various other properties, like version number of our project etc.

A terminal window with a dark background and three colored window control buttons (red, yellow, green) at the top left. It displays a JSON object representing a package.json file. The lines are numbered 1 through 16 on the left side of the terminal.

```
1  {
2    "name": "npm_basics",
3    "version": "1.0.0",
4    "description": "",
5    "main": "index.js",
6    "scripts": {
7      "test": "echo \"Error: no test specified\" && exit 1"
8    },
9    "keywords": [],
10   "author": "",
11   "license": "ISC",
12   "dependencies": {
13     "lodash": "^4.17.21"
14   }
15 }
16
```

- Notice the “dependencies” object, it has the list of installed packages.
- Note `npm install lodash` and `npm i lodash` are same.
- If you run install the same package again and again, npm doesn't download a new copy every time.

## Remove a package

```
>npm remove <name-of-the-package>
```

## Global Packages

- When we run `npm install lodash`, it doesn't modify any file outside of the directory we are working in.
- Such packages are said to be downloaded locally.
- But some node packages have features that they require the access of command line, these can be downloaded globally.

```
>npm install nodemon -g
```

- Or

```
>npm install nodemon --global
```

- We can then access these modules without using NPM

```
>nodemon app.js
```

- We can check where these global modules are stored on our computer using

```
>npm root -g
```

- Interestingly, NPM is also a globally installed package.
- Hence we can update NPM using

```
>npm install -g npm@latest
```

- NPM uses, semantic versioning to manager its packages, if you are not familiar with it, read here:

[https://s3-us-west-2.amazonaws.com/secure.notion-static.com/cb08abb3-3ab0-4896-9848-72a29e0d59c7/Semantic\\_Versioning.pdf](https://s3-us-west-2.amazonaws.com/secure.notion-static.com/cb08abb3-3ab0-4896-9848-72a29e0d59c7/Semantic_Versioning.pdf)

- NPM has no way of enforcing semantic version. There is no guarantee that a patch in the dependency won't break your app.

## npm install

- If we remove the node\_modules folder from the directory, and then run

```
>npm install
```

- NPM will install all the required packages mentioned in the package.json file.

## package-lock.json

Note: This may get tricky to understand, watch the video for this section first.

- As we discussed NPM uses semantic versioning but has no way to enforce it, that means if we update a package from 2.1.3 to 2.2.0, there is no guarantee that it won't break our app. In an ideal world, it shouldn't break, but, it's not an ideal world.
- So when we do `npm install`, and NPM sees that we are using version ^2.3.4 (notice the caret symbol) of a software then NPM checks if it can get the updated version of the software without breaking the app, that is the highest minor\_change number, something like, 2.17.0. But there is no guarantee.
- There can be another case, that the dependency of our package has changed, and these changes are breaking.
- For the same reasons, NPM manages the package-lock.json file, this file stores the version of the package that we installed (while development) and the version of all the dependencies for that version at that point of time.
- So the time when we run `npm install`, NPM installs only those version.

```


1  {
2    "name": "npm_basics",
3    "version": "1.0.0",
4    "lockfileVersion": 2,
5    "requires": true,
6    "packages": {
7      "": {
8        "name": "npm_basics",
9        "version": "1.0.0",
10       "license": "ISC",
11       "dependencies": {
12         "lodash": "^4.17.21"
13       }
14     },
15     "node_modules/lodash": {
16       "version": "4.17.21",
17       "resolved": "https://registry.npmjs.org/lodash/-/lodash-4.17.21.tgz",
18       "integrity": "sha512-v2kDEe571ecTulaDIuNTPy3Ry4gLGJ6Z103vE1krgXZNrsQ+LFTGHVxVjcXPs17LhbZVGedAJv8XZ1tvj5FvSg=="
19     }
20   },
21   "dependencies": {
22     "lodash": {
23       "version": "4.17.21",
24       "resolved": "https://registry.npmjs.org/lodash/-/lodash-4.17.21.tgz",
25       "integrity": "sha512-v2kDEe571ecTulaDIuNTPy3Ry4gLGJ6Z103vE1krgXZNrsQ+LFTGHVxVjcXPs17LhbZVGedAJv8XZ1tvj5FvSg=="
26     }
27   }
28 }
29

```

- There was no package-lock.json file before NPM version 5, the above mentioned problems led to the creation of this file.

## dependencies

- All the packages installed go to the dependencies object of the package.json file



```
1  "dependencies": {  
2    "lodash": "^4.17.21"  
3  }
```

- When we run `npm install`, NPM install the packages mentioned in this object.

## devDependencies

- A lot times we use packages that help us during the development process, like babel, webpack etc.
- These packages are not required to run our application.
- For example if we are using the package lodash and webpack, our app would crash without lodash but not without webpack(because it is only required during development).
- Now to segregate our devDependencies we can install them using the `—save-dev` flag

```
>npm install webpack --save-dev
```

- This will save our package to a different section named devDependencies





```

1  {
2    "name": "npm_basics",
3    "version": "1.0.0",
4    "description": "",
5    "main": "index.js",
6    "scripts": {
7      "test": "echo \"Error: no test specified\" && exit 1"
8    },
9    "keywords": [],
10   "author": "",
11   "license": "ISC",
12   "dependencies": {
13     "lodash": "^4.17.21"
14   },
15   "devDependencies": {
16     "webpack": "^5.70.0"
17   }
18 }

```

- Now whenever we run

```
>NODE_ENV="production" npm install
```

- That is if our application is running in production, it will only download the dependencies and not devDependencies.

## peerDependencies

Note: This section is not super important

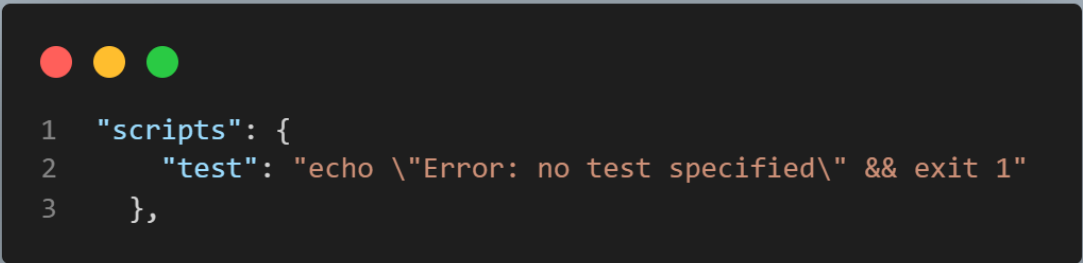
- These dependencies are mostly used by package developers, and not by others.
- For example if we do

```
>npm install react-dom
```

- If would make no sense, since react-dom need react to work, the package developer for react-dom would mention in package.json file of the package that it requires react to work, this information would go into the peerDependency section. So when we install react-dom, npm would throw a warning.

## NPM Scripts

- If we see our package.json file, there is an object called scripts.

A terminal window with a dark background and three colored window control buttons (red, yellow, green) at the top left. It displays a snippet of a package.json file's 'scripts' object.

```
1  "scripts": {  
2    "test": "echo \"Error: no test specified\" && exit 1"  
3  },
```

- Like all objects it has key value pair.
- The key can be anything, the value is a command line script.
- It can be node or python script as well.
- Now NPM comes with special script keys like **prepublish** **prepare** etc.
- These are called life-cycle scripts. They run before and after certain events occur like preinstall runs before a package is installed
- We can write these scripts into the script object in package.json



```
1  "scripts": {  
2    "test": "echo \"Error: no test specified\" && exit 1",  
3    "start": "echo Hello World"  
4  },
```

- We can then run these scripts(predefined) by

```
>npm start
```

- But this does not mean that we can write `npm <some-script-name>` for any script. It works for predefined scripts only.
- For scripts that are not predefined, we need to use `run` command.



```
1  "scripts": {  
2    "test": "echo \"Error: no test specified\" && exit 1",  
3    "start": "echo Hello World",  
4    "hello": "echo npm says hello"  
5  },
```

```
>npm run hello
```

## NPX

- Was introduced in version 5.2.0
- NPX allows to run npm tools without installing them globally.
- For example, to run create-react-app, we need to first install it globally

```
>npm install create-react-app -g
```

- And then

```
>create-react-app myApp
```

- Using npx we can use create-react-app without installing it globally

```
>npx create-react-app myApp
```

## Changing global package installation location

- Run the following command to see the location where global packages are saved

```
>npm root -g
```

- Run the following command to change the location where global packages will be installed

```
>npm config set prefix <path-to-be-set>

//example
>npm config set prefix /Users/dev/npm_basics/abc
```

## Caching and Purging

- NPM keeps a hidden folder .npm that caches the packages.
- So if we run npm install and there is already a package with same name and version , NPM doesn't make network request for it. It simply returns the package from the cache.
- We can clear the cache in case NPM is not making network calls and we want it to do the same.

```
>npm cache clean --force
```