**03 Node Package Manager**

**1) NPM (Node Package Manager)**:

NPM (short for Node.js package manager) is a package manager for the JavaScript programming language. It is the default package manager for the JavaScript runtime environment Node.js. It consists of a command line client, also called NPM, and an online database of public and paid-for private packages, called the NPM registry. The registry is accessed via the client, and the available packages can be browsed and searched via the NPM website. The package manager and the registry are managed by NPM, Inc.

NPM is used to manage dependencies for packages. If we were to unpack a framework and use it outside NPM, we would have to do this every time we want to update the framework. NPM does this for us. We always know what version we are on, and we can limit a dependency to a specific major/minor/patch version.

**In this section we learn**:

1. Every Node application has a package.json file that includes metadata about the application. This includes the name of the application, its version, dependencies, etc.
2. We use NPM to download and install 3rd-party packages from NPM registry:
3. All the installed packages and their dependencies are stored under node modules folders. This folder should be excluded from the source control.
4. Node packages follow semantic versioning: major.minor.patch

**Useful NPM commands are**:

1. Install a package

npm i <packageName> (Specific for application)

npm i -g <packageName> (Install globally)

sudo npm i -g <packageName> (For Linux)

1. Check NPM version

npm -v

1. Install a specific version of a package

npm i <packageName>@<version> (Specific for application)

npm i -g <packageName>@<version> (Install globally)

sudo npm i -g <packageName>@<version> (For Linux)

1. Install a package as a development dependency

npm i <packageName> —save-dev

1. Uninstall a package

npm un <packageName>

1. List installed packages

npm list —depth=0 (in console only module)

npm list (Everywhere (VS code terminal) all version )

1. View outdated packages

npm outdated

1. View outdated packages

npm update

**2) Package.json**:

All NPM packages contain a file, usually in the project root, called package.json - this file holds various metadata relevant to the project. This file is used to give information to NPM that allows it to identify the project as well as handle the project's dependencies. It can also contain other metadata such as a project description, the version of the project in a particular distribution, license information, even configuration data - all of which can be vital to both NPM and to the end users of the package. The package.json file is normally located at the root directory of a Node.js project.

**Create package.json**:

Go to the project directory and open CMD and run the following command

1. npm init
2. Press ^C at any time to quit.

package name: (practice) (place name or press enter)

1. version: (1.0.0) (Press Enter)
2. description: (Press Enter)
3. entry point: (index.js) (Press Enter)
4. test command: (Press Enter)
5. git repository: (Press Enter)
6. keywords: (Press Enter)
7. author: (Press Enter)
8. license: (ISC) (Press Enter)
9. Is this ok? (yes) (Press Enter)

Now "package.json" file is created in our directory.

**package.json**:

{

"name": "practice",

"version": "1.0.0",

"description": "",

"main": "index.js",

"scripts": {

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

},

"author": "",

"license": "ISC"

}

There is a faster way to create "package.json" file. Go to cmd and run "npm init --yes"

**3) Installing a Node Package**:

Now we add a third-party library or a third-party node package to our node application. We install a popular JavaScript library called Underscore.

npm i underscore

Now this package is added in our project. Now two things happened in our project.

1. In package.json added a new property name "dependencies"
2. Under that underscore package is added "underscore": "^1.9.1"

{

"name": "practice",

"version": "1.0.0",

"main": "index.js",

"scripts": {

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

},

"author": "",

"license": "ISC",

"keywords": [],

"description": "",

"dependencies": {

"underscore": "^1.9.1"

}

}

In package.json we specify all the dependencies our project and there version. When we are using "npm install", npm is going to download the latest version of the given package from npm registry and then it will store it inside a folder called "node\_modules" in our current working directory.

Previous we use the following command to add dependency in package.json file

npm i underscore --save

**4) Using a Package**:

In the root directory of the project create a file named "index.js" and write the following code.

**index.js**:

var \_ = require("underscore");

var result = \_.contains([1, 2, 3], 2);

console.log(result); *//true*

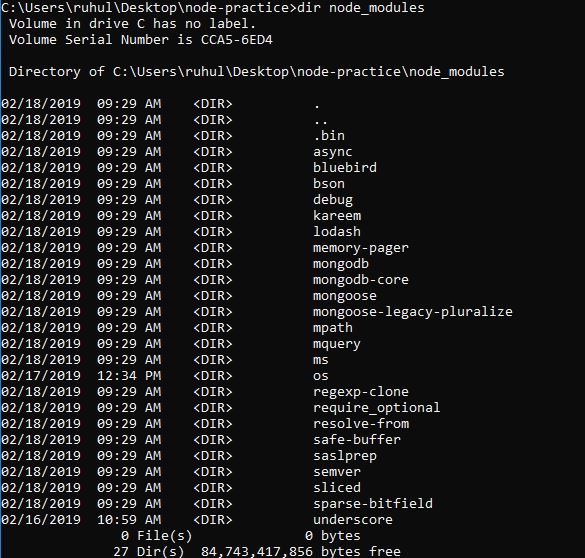
Now for run this code goto console and run the following command.

node index.js

**5) Package Dependencies**:

Inside node module we can see what we have, by using the following command.

dir node\_modules



Here we see a lot of packages, but we don’t install all of those packages. We install only mongoose, os, and underscore and the remaining packages are installed because the three packages are depending on those packages.

In previous all the dependencies of a given package are install inside the package folder. But now all dependencies are installing inside "node\_modules" folder. If one of the packages install a different version, that version will be stored locally in the package.

**6) NPM Packages and Source Control**:

In a real-world application we are going to have lots of folder inside "node\_modules" folder and the size of "node\_modules" folder will grow significantly. The side of "node\_modules" may be a few hundred MB. When we upload our source code into a source control repository, we don’t upload the "node\_modules" in the repository. The same is true when we copy our code from machine to machine.

All our dependencies are store inside "package.json" file. We can easily restore this version of this dependencies in any machine.

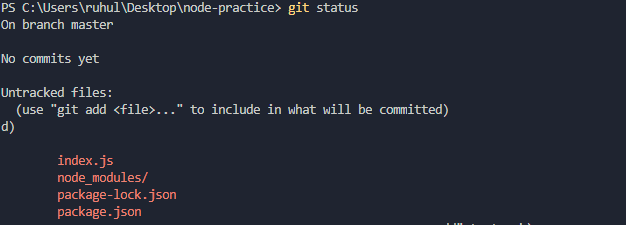
Remove "node\_modules" from our project. Back in the terminal and run the following command.

npm install

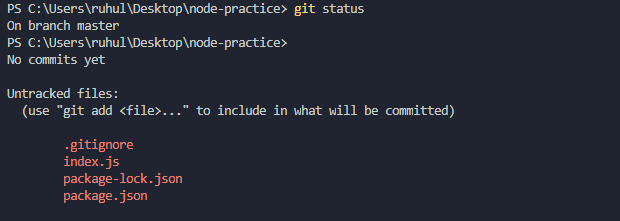
Now it will download all dependencies from NPM registry.

If we keep our project in git we have to do the following work for not upload the "node\_modules" inside git repository.

1. Back in the terminal
2. git init
3. git status (We will see the file that we add to our repository).



1. For not upload "node\_modules" back in our project root directory.
2. In the root add a file name ".gitignore".
3. Now in this file we will list out the file that we don’t want to upload in git.
4. Go to ".gitignore" and write "node\_modules/" and save, here / indicate that it is a folder.
5. Back in the terminal and run git status again.



1. Run git commit -m "Our first commit"

**7) Semantic Versioning**:

Semantic Versioning is a convention used to provide a meaning to versions. If there’s one great thing in Node.js packages, is that all agreed on using Semantic Versioning for their version numbering.

The Semantic Versioning concept is simple: all versions have 3 digits: x.y.z. (Major.Minor.Patch)

* The first digit is the major version
* The second digit is the minor version
* The third digit is the patch version

When we make a new release, we don’t just up a number as we please, but we have rules:

* We up the major version when we make incompatible API changes
* We up the minor version when we add functionality in a backward-compatible manner but don’t break the existing API
* We up the patch version when we make backward-compatible bug fixes

The convention is adopted all across programming languages, and it is very important that every NPM package adheres to it, because the whole system depends on that.

**Why is that so important**:

Because NPM set some rules we can use in the "package.json" file to choose which versions it can update our packages to, when we run NPM update. (Reference: <https://flaviocopes.com/npm-semantic-versioning/> )

|  |  |
| --- | --- |
| Sign | Meaning |
| **^** | if we write ^0.13.0 when running "NPM update" it can update to patch and minor releases: 0.13.1, 0.14.0 and so on. |
| **~** | if we write ~0.13.0, when running "NPM update" it can update to patch releases: 0.13.1 is ok, but 0.14.0 is not. |
| **>** | We accept any version higher than the one we specify. |
| **>=** | We accept any version equal to or higher than to the one we specify |
| **<** | We accept any version lower to the one we specify. |
| **<=** | We accept any version equal to or lower to than the one we specify |
| **=** | We accept the exact version |
| **-** | We accept a range of versions. Example: (2.1.0 - 2.6.2) |
| **||** | We combine sets. Example: (< 2.1 || > 2.6) |

You can combine some of those notations, for example use 1.0.0 || >=1.1.0 <1.2.0 to either use 1.0.0 or one release from 1.1.0 up, but lower than 1.2.0.

There are other rules, too:

* no symbol: you accept only that specific version you specify (1.2.1)
* latest: you want to use the latest version available

**8) Listing the Installed Packages**:

We can know the exact version of our packages install in our node application in two ways.

* "package.json" inside the package folder
* By using command in console

**Inside package**:

Go to the project directory -> package\_folder -> package.json. For example if we want to know the version of underscore package we do the following thing

Node\_module -> underscore -> package.json

"version": "1.9.1"

But this is not a smart way. We can easily do this by using command

**By using command in console**:

List installed packages

npm list —depth=0 (in console only module)

npm list (Everywhere (VS code terminal) all version )

**9) Viewing Registry Info for a Package**:

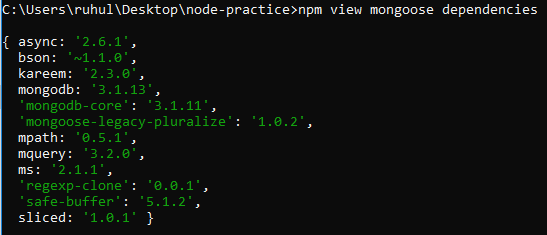
If we want to learn about an NPM package we have to go the link "<https://www.npmjs.com>". From the search box we can find any package and can see metadata like versions, dependencies etc.

We can know details about the package from our console. We have to run the following command

npm view mongoose

we see the details info about mongoose. But it provides us a lot of info. We get only dependences info by using the following command.

npm view mongoose dependencies



We can also see all the versions of a package by using the following command.

npm view mongoose versions

**10) Installing a Specific Version of a Package**:

We can install a specific version of package by using the following command.

npm i mongoose@5.4.10 (install mongoose 5.4.10 version)

npm i underscore@1.4.0 (install underscore 1.4.0 version)

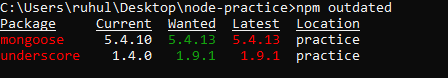
For see the package version run the following command

npm list --dept=0

**11) Updating Local Packages**:

To see which version of package is outdate we can run the following command

npm outdated



Here we see the version of the packages install in our project, wanted versions and the latest version. Now for update the outdated version run the following command.

npm update (this only works for minor and patch version)

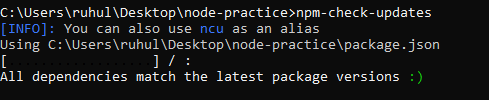
The update package installed in our project.

If we want to update the very latest version, we have to install a different command line tool. For this we have to run the following command.

npm i -g npm-check-updates

Now we have a new command line tool name "npm-check-updates". For all outdated packages and their new version run

npm-check-updates



If any package need update, we have to run

ncu -u

Now the package is only update not install in our project. For install the dependencies we have to go console and run

npm i

**12) Dev-Dependencies**:

Development dependencies are intended as development-only packages, that are unneeded in production. For example, testing packages, webpack or Babel.

Here we install JSHint package as Dev-Dependencies. JSHint is a static code analysis tool used in software development for checking if JavaScript source code complies with coding rules.

For add JSHint as Dev-Dependencies run the following command.

npm i jshint --save-dev

{

"name": "practice",

"version": "1.0.0",

"main": "index.js",

"scripts": {

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

},

"author": "",

"license": "ISC",

"keywords": [],

"description": "",

"dependencies": {

"mongoose": "^5.4.13",

"os": "^0.1.1",

"underscore": "^1.9.1"

},

"devDependencies": {

"jshint": "^2.10.1"

}

}

We have to add an extra --save-dev flag for Dev-Dependencies.

**13) Uninstalling a Package**:

In the real-world application sometime, we may need to uninstall a package. For this we have to run the following command.

npm un mongoose

Now mongoose module is uninstalled from our project.

{

"name": "practice",

"version": "1.0.0",

"main": "index.js",

"scripts": {

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

},

"author": "",

"license": "ISC",

"keywords": [],

"description": "",

"dependencies": {

"os": "^0.1.1",

"underscore": "^1.9.1"

},

"devDependencies": {

"jshint": "^2.10.1"

}

}

This is removing from the node-module folder also.

**14) Working with Global Packages**:

All this package we are install in our project like underscore, os, mongoose is particular to this node

Project in the folder. But there are node packages on npm registry that are not specific to an application. This are often command line tools that we want to access from everywhere. They are not for a specific folder or a specific project.

For example, NPM is an example of a global package. It’s a command line tool that we can run from anywhere in our machine.

We can globally install a package by using an extra -g flag. For install NPM globally we have to run the following command.

npm i -g npm

We can check the outdated global npm package by using the following command

npm -g outdated



**15) Publishing a Package**:

We can create our own npm package and publish it to the npm registry. For this we have to do the following thing.

1. Create a new folder

mkdir idb-bisew

1. Go to the folder

Cd idb-lib

1. Make an index.json file inside idb-lib

npm init –yes

1. Open the folder in vscode (set path variable first)

code .

1. Add a new file name index.js and write the following code

module.exports.add = function(a, b) {

return a + b;

};

1. Go to console and login npm
   1. Username: your\_username
   2. Password: your\_password
   3. Email: your\_email\_id

If login successfully we will see the message

Logged in as ruhulmath08 on <https://registry.npmjs.org/>.

1. Now for publish our package run

npm publish



Now we can use it in another node application.

**Use package**:

Now for using our package we have to do

1. Go to a new project

cd node-practice

1. Install the package

npm i idb-lib-1

1. Create a file named index.js in root directory
2. Load the package and use it

const idb = require("idb-lib-1");

let result = idb.add(10, 20);

console.log(result); *//30*

**16) Updating a Published Package**:

Now suppose we add a new function in the package idb-lib-1

1. Go to idb-lib

cd idb-lib

1. Write the function

module.exports.add = function(a, b) {

return a + b;

};

*//new added function*

module.exports.multiply = function(a, b) {

return a \* b;

};

1. Go to package.json and update the version by hand or run the command

npm version minor

1. Now publish the package

npm publish

Node Package Manager