Day 0/1

Monday, September 7, 2020 2:42 PM

A package is a collection of files and configuration wraps in an easy to distribute wrapper.
Any package that your application relies on to work is referred to as a dependency.
Package managers are applications that accept your code bundled up with some important meta-data and provide services like versioning, change management and even tracking how many projects are using your code.
Most package managers consist of at least two parts, a command line interface, and a registry.
NPM stands for node package manager.

npm will show npm's help info, including some common commands and how to access more detailed guides.
npm init will set your current project directory up for npm.
This requires answering a few questions to generate a package.jsonfile,
a critical part of npm's dependency management functionality.npm install will download and install a package into your project.
You canuse the-g(or--global) flag to install a package for use everywhere on your system.
To have some fun, run npm install -g cowsay.
Once thedownload's completed, try runningcowsay Hello, world

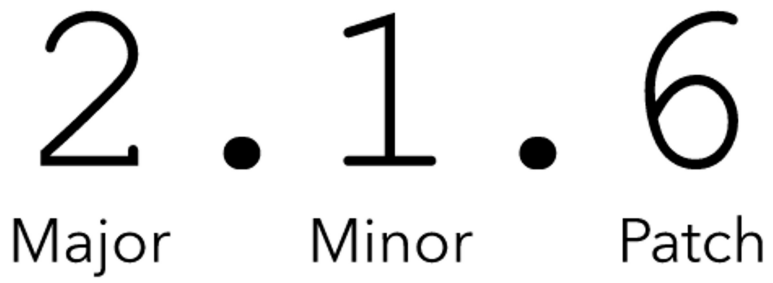
Mini package managers including NPM have the ability to resolve correct dependency versions, this means the manager can compare all the packages used by an application and determine which versions are most compatible.
NPM accomplishes this dependency resolution process using both the package.json and package-lock.json files.
Package-lock.json is commonly known as a lock file in package manager parlance.
When the NPM CLI utility installs a package it adds to the node modules subdirectory in your project.
Each package will be placed in a directory named after itself and contain the role code for the package along with any associated package.Json as well as documentation.
The node_modules file is special for a few reasons:
It’s a great way to keep dependencies separated for each project.
By keeping node modules for each project separate you can have as many different versions of each package as you like each project has a specific version it needs right on hand.

Because of the amount of space they take up it is good practice to keep node modules out of get repository's or other version control.

Semantic.version.number (Often abbreviated as semvar) : is a way of tracking version numbers that lets other developers know what to expect from each release of your package.

Semantic version numbers are made up of three parts each numbered sequentially in with no limit on how large they can be, the leftmost digit in a semvar is the most significant meaning that
1.0.0 > 0.8.99

Here's a high-level overview:



Major changes should be considered breaking.
They will be incompatible with other major versions of the same package and may require significant changes in any app that depends on them. Creating a sequel to a hit video game would be a major change.

Minor changes generally represent new features. These shouldn't totally break anything, but might require a little tweak to keep dependent apps up-to-date. Adding a new level to a video game would be a minor change.

Patch-level changes are for fixing bugs or small issues. These shouldn't break any other functionality or force dependent apps to make any changes themselves. Fixing a typo in a video game's instructions would be a patch-level change

When adding a new dependency to your package.jsonfile, you can designate a range by adding some special characters to your version number:

- *indicates "whatever the latest version is".>
- 1.0.0indicates "any version above major version
- 1".^1.0.0indicates "any version in the 1.x.x range".
- ~1.0.0indicates "any patch version in the 1.0.x range"
- .1.0.0indicates "exactly version 1.0.0".

You may also omit consecutive trailing zeroes, so:
^1.0.0 is the same as ^1.0 or just ^1.~1.0.2----------("any patch version greater than 1.0.2") would need to written out in its entirely, though.
You should consider the numbers in your semver to be a minimum value, ----------so~2.1.3 would include2.1.4, but not2.1.2.

while npm helps manage your dependencies, it won't automatically keep them up to date!

Using npm to Perform Common Tasks -Part One:

Using npm to manage npm :

To confirm if you have the npm CLI installed, you can run the command npm --version
Or npm -v.

If you have the npm CLI installed, you'll see its version number displayed in the console.

If you don't have the npm CLI installed, you receive an error.

The npm CLI is available as an npm package, which allows you to use the npm CLI to update itself.

If you're using macOS or Linux, you can update the npm CLI to the latest version by running the following command :
npm install -g npm@latest

If you installed Node.js using the default installer, you might need to prefix the above command with sudo like this:

sudo npm install -g npm@latest

The sudo command allows you to run a command with the security privileges of another user, typically your computer's administrator or super user account.

When using the sudo command, you'll be prompted for your account's password.

Any node.js project that contains a package.json file is technically a NPM package the most of these projects will never be published at the NPM registry for consumption by the general development community.

When selecting a package, it's helpful to ask yourself the following questions:

- **Does the package do what I need?**Most packages in the npm registry will include some documentation on how to use the package. Usually you can review that documentation to determine if the package will suit your needs. Sometimes, you might need to review any additional documentation that's available in the package's code repository (i.e. GitHub, GitLab, or wherever the package's source code lives). Alternatively, you can install the package into a throwaway project to safely test it in a sandbox environment.
- **How popular is the package?**Popularity isn't always everything, but it can be an effective way to determine if a package is useful and reliable. It also increases the likelihood that other developers on your team might have experience with using a particular package.
- **Is the package being maintained?**If you're going to take a dependency on a package, you need to have confidence that the package is actively being maintained by its developer(s). To do that, review the package's associated code repository (i.e. GitHub, GitLab, etc.) Have there been recent commits and recent releases? Review the repository's issues to see if consumers of the package are getting their questions answered and bugs are being fixed.

Using a dependency in code:

Dependency type:

npm tracks two types of dependencies in the package.jsonfile:

1. Dependencies (dependencies) - These are the packages that your project needs in order to successfully run when in production (i.e. your application has been deployed or published to a server that can be accessed by your users).
2. Development dependencies (devDependencies) - These are the packages that are needed locally when doing development work on the project. Development dependencies often include one or more tools that are used to build and test your application.

➢ There are actually three additional types dependencies that npm can track

a. including peer dependencies (peerDependencies),
b. bundled dependencies(bundledDependencies), and
c. optional dependencies(optionalDependencies).

Installing a development dependency:

To install a development dependency, you simply add the-`save-dev flag:
npm install mocha --save-dev

The--save-devflag causes npm to add the package to the devDependencies field in the package.jsonfile:
{"dependencies":{"colors":"^1.4.0"},

Installing a development dependency

To install a development dependency, you simply add the `--save-dev` flag:

```
npm install mocha --save-dev
```

The `--save-dev` flag causes npm will add the package to the `devDependencies` field in the `package.json` file:

```
{
  "dependencies": {
    "colors": "^1.4.0"
  },
```

```
  "devDependencies": {
    "mocha": "^7.0.1"
  }
}
```

Separating the development dependencies from the application's "regular" dependencies keeps the package installation process as lean as possible, by allowing npm to install just the packages that are actually needed for the package or application to successfully run.

Using npm to Perform Common Tasks -Part Two:

When getting started with an existing project that already contains package.json and package-lock.json files, you will need to use NPM to install dependencies if you don't install them you will receive errors when attempting to run the application.

to install an existing projects dependencies simply run the NPM install command, this causes NPM to install the dependencies listed in the package lock file.

Uninstalling a dependency:

Imagine that you install the `lodash` package by running the following command:

```
npm install lodash
```

Here's the output displayed by npm after the package is installed:

```
+ lodash@4.17.15
added 1 package from 2 contributors and audited 1 package in 0.609s
found 0 vulnerabilities
```

At this point, your `node_modules` folder will contain a folder named `lodash` that contains the code for the `lodash` package. Your `package.json` file will also list `lodash` as a dependency:

```
{
  "dependencies": {
    "lodash": "^4.17.15"
  }
}
```

To remove or uninstall the load dash package you can run the following command

npm uninstall load ash

When NPM uninstall the package it will remove the packaging all of its dependencies from the nose modules folder it will also update the package that Jason file in the package lock file so the package is no longer listed as a dependency.

Updating a dependency

Imagine that you added the `lodash` package as a dependency to a project awhile ago; back when the latest version of `lodash` was `3.0.0`. You can simulate this situation by installing a specific version of `lodash`:

```
npm install lodash@3.0.0
```

Which results in the following output:

```
+ lodash@3.0.0
added 1 package from 5 contributors and audited 1 package in 0.606s
found 3 vulnerabilities (1 low, 2 high)
  run `npm audit fix` to fix them, or `npm audit` for details
```

For now, ignore the security vulnerabilities that npm found. You'll learn in a bit how to use `npm audit` to resolve those issues.

Here's what the dependency in the `package.json` file looks like at this point:

```
{
  "dependencies": {
    "lodash": "^3.0.0"
  }
}
```

Here's what the dependency in the `package.json` file looks like at this point:

```
{
  "dependencies": {
    "lodash": "^3.0.0"
  }
}
```

Remember that server ^3.0.0 Means that you'll except any minor and patch versions For loadash major version three

If you want to update lodash, so you can use some features that were added in version3.1.0.
You can update the lodash packageby running the following command:
npm update lodash

And here's what your dependencies look like in the `package.json` file:

```
{
  "dependencies": {
    "lodash": "^3.10.1"
  }
}
```

Updating all project dependencies: