npm-package-locks

An explanation of npm lockfiles

DESCRIPTION

Conceptually, the “input” to npm-install is a package.json, while its “output” is a fully-formed `node_modules` tree: a representation of the dependencies you declared. In an ideal world, npm would work like a pure function: the same `package.json` should produce the exact same `node_modules` tree, any time. In some cases, this is indeed true. But in many others, npm is unable to do this. There are multiple reasons for this:

- different versions of npm (or other package managers) may have been used to install a package, each using slightly different installation algorithms.
- a new version of a direct semver-range package may have been published since the last time your packages were installed, and thus a newer version will be used.
- A dependency of one of your dependencies may have published a new version, which will update even if you used pinned dependency specifiers (`1.2.3` instead of `^1.2.3`)
- The registry you installed from is no longer available, or allows mutation of versions (unlike the primary npm registry), and a different version of a package exists under the same version number now.

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.

To prevent this potential issue, npm uses package-lock.json or, if present, npm-shrinkwrap.json. These files are called package locks, or lockfiles.

Whenever you run **`npm install`**, npm generates or updates your package lock, which will look something like this:

```
{
  "name": "A",
  "version": "0.1.0",
  ...metadata fields...
  "dependencies": {
```

```
    "B": {
      "version": "0.0.1",
      "resolved": "https://registry.npmjs.org/B/-/B-0.0.1.tgz",
      "integrity": "sha512-DeAdb33F+"
      "dependencies": {
        "C": {
          "version": "git://github.com/org/C.git#5c380ae319fc4efe9e7f2d9c78b0faa5
        }
      }
    }
  }
}
```

This file describes an *exact*, and more importantly *reproducible* `node_modules` tree. Once it's present, any future installation will base its work off this file, instead of recalculating dependency versions off package.json.

The presence of a package lock changes the installation behavior such that:

1. The module tree described by the package lock 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.

If `preshrinkwrap`, `shrinkwrap` or `postshrinkwrap` are in the `scripts` property of the `package.json`, they will be executed in order. `preshrinkwrap` and `shrinkwrap` are executed before the shrinkwrap, `postshrinkwrap` is executed afterwards. These scripts run for both `package-lock.json` and `npm-shrinkwrap.json`. For example to run some postprocessing on the generated file:

```
"scripts": {
  "postshrinkwrap": "json -I -e \"this.myMetadata = $MY_APP_METADATA\""
}
```

Using locked packages

Using a locked package is no different than using any package without a package lock: any commands that update `node_modules` and/or `package.json`'s dependencies will automatically sync the existing lockfile. This includes `npm install`, `npm rm`, `npm update`, etc. To prevent this update from happening, you can use the `--no-save` option to prevent saving altogether, or `--no-`

`shrinkwrap` to allow `package.json` to be updated while leaving `package-lock.json` or `npm-shrinkwrap.json` intact.

It is highly recommended you commit the generated package lock to source control: this will allow anyone else on your team, your deployments, your CI/continuous integration, and anyone else who runs `npm install` in your package source to get the exact same dependency tree that you were developing on. Additionally, the diffs from these changes are human-readable and will inform you of any changes npm has made to your `node_modules`, so you can notice if any transitive dependencies were updated, hoisted, etc.

Resolving lockfile conflicts

Occasionally, two separate npm install will create package locks that cause merge conflicts in source control systems. As of `npm@5.7.0`, these conflicts can be resolved by manually fixing any `package.json` conflicts, and then running `npm install [--package-lock-only]` again. npm will automatically resolve any conflicts for you and write a merged package lock that includes all the dependencies from both branches in a reasonable tree. If `--package-lock-only` is provided, it will do this without also modifying your local `node_modules/`.

To make this process seamless on git, consider installing `npm-merge-driver`, which will teach git how to do this itself without any user interaction. In short: `$ npx npm-merge-driver install -g` will let you do this, and even works with pre-`npm@5.7.0` versions of npm 5, albeit a bit more noisily. Note that if `package.json` itself conflicts, you will have to resolve that by hand and run `npm install` manually, even with the merge driver.