package-lock.json

Description

`package-lock.json` is automatically generated for any operations where npm modifies either the `node_modules` tree, or `package.json`. It describes the exact tree that was generated, such that subsequent installs are able to generate identical trees, regardless of intermediate dependency updates.

This file is intended to be committed into source repositories, and serves various purposes:

- Describe a single representation of a dependency tree such that teammates, deployments, and continuous integration are guaranteed to install exactly the same dependencies.
- Provide a facility for users to "time-travel" to previous states of `node_modules` without having to commit the directory itself.
- Facilitate greater visibility of tree changes through readable source control diffs.
- Optimize the installation process by allowing npm to skip repeated metadata resolutions for previously-installed packages.
- As of npm v7, lockfiles include enough information to gain a complete picture of the package tree, reducing the need to read `package.json` files, and allowing for significant performance improvements.

`package-lock.json` vs `npm-shrinkwrap.json`

Both of these files have the same format, and perform similar functions in the root of a project.

The difference is that `package-lock.json` cannot be published, and it will be ignored if found in any place other than the root project.

In contrast, npm-shrinkwrap.json allows publication, and defines the dependency tree from the point encountered. This is not recommended unless deploying a CLI tool or otherwise using the publication process for producing production packages.

If both `package-lock.json` and `npm-shrinkwrap.json` are present in the root of a project, `npm-shrinkwrap.json` will take precedence and `package-lock.json` will be ignored.

Hidden Lockfiles

In order to avoid processing the `node_modules` folder repeatedly, npm as of v7 uses a "hidden" lockfile present in `node_modules/.package-lock.json`. This contains information about the tree, and is used in lieu of reading the entire `node_modules` hierarchy provided that the following conditions are met:

- All package folders it references exist in the `node_modules` hierarchy.
- No package folders exist in the `node_modules` hierarchy that are not listed in the lockfile.
- The modified time of the file is at least as recent as all of the package folders it references.

That is, the hidden lockfile will only be relevant if it was created as part of the most recent update to the package tree. If another CLI mutates the tree in any way, this will be detected, and the hidden lockfile will be ignored.

Note that it *is* possible to manually change the *contents* of a package in such a way that the modified time of the package folder is unaffected. For example, if you add a file to `node_modules/foo/lib/bar.js`, then the modified time on `node_modules/foo` will not reflect this change. If you are manually editing files in `node_modules`, it is generally best to delete the file at `node_modules/.package-lock.json`.

As the hidden lockfile is ignored by older npm versions, it does not contain the backwards compatibility affordances present in "normal" lockfiles. That is, it is `lockfileVersion: 3`, rather than `lockfileVersion: 2`.

Handling Old Lockfiles

When npm detects a lockfile from npm v6 or before during the package installation process, it is automatically updated to fetch missing information from either the `node_modules` tree or (in the case of empty `node_modules` trees or very old lockfile formats) the npm registry.

File Format

`name` The name of the package this is a package-lock for. This will match what's in `package.json`.

`version` The version of the package this is a package-lock for. This will match what's in `package.json`.

`lockfileVersion` An integer version, starting at `1` with the version number of this document whose semantics were used when generating this `package-lock.json`.

Note that the file format changed significantly in npm v7 to track information that would have otherwise required looking in `node_modules` or the npm registry. Lockfiles generated by npm v7 will contain `lockfileVersion: 2`.

- No version provided: an "ancient" shrinkwrap file from a version of npm prior to npm v5.
- `1`: The lockfile version used by npm v5 and v6.
- `2`: The lockfile version used by npm v7, which is backwards compatible to v1 lockfiles.
- `3`: The lockfile version used by npm v7, *without* backwards compatibility affordances. This is used for the hidden lockfile at `node_modules/.package-lock.json`, and will likely be used in a future version of npm, once support for npm v6 is no longer relevant.

npm will always attempt to get whatever data it can out of a lockfile, even if it is not a version that it was designed to support.

`packages` This is an object that maps package locations to an object containing the information about that package.

The root project is typically listed with a key of `""`, and all other packages are listed with their relative paths from the root project folder.

Package descriptors have the following fields:

- version: The version found in `package.json`
- resolved: The place where the package was actually resolved from. In the case of packages fetched from the registry, this will be a url to a tarball. In the case of git dependencies, this will be the full git url with commit sha. In the case of link dependencies, this will be the location of the link target. `registry.npmjs.org` is a magic value meaning "the currently configured registry".
- integrity: A `sha512` or `sha1` Standard Subresource Integrity string for the artifact that was unpacked in this location.
- link: A flag to indicate that this is a symbolic link. If this is present, no other fields are specified, since the link target will also be included in the lockfile.
- dev, optional, devOptional: If the package is strictly part of the `devDependencies` tree, then `dev` will be true. If it is strictly part of the `optionalDependencies` tree, then `optional` will be set. If it is both a `dev` dependency *and* an `optional` dependency of a non-dev dependency,

then `devOptional` will be set. (An `optional` dependency of a `dev` dependency will have both `dev` and `optional` set.)

- inBundle: A flag to indicate that the package is a bundled dependency.
- hasInstallScript: A flag to indicate that the package has a `preinstall`, `install`, or `postinstall` script.
- hasShrinkwrap: A flag to indicate that the package has an `npm-shrinkwrap.json` file.
- bin, license, engines, dependencies, optionalDependencies: fields from `package.json`

dependencies Legacy data for supporting versions of npm that use `lockfileVersion: 1`. This is a mapping of package names to dependency objects. Because the object structure is strictly hierarchical, symbolic link dependencies are somewhat challenging to represent in some cases.

npm v7 ignores this section entirely if a `packages` section is present, but does keep it up to date in order to support switching between npm v6 and npm v7.

Dependency objects have the following fields:

- version: a specifier that varies depending on the nature of the package, and is usable in fetching a new copy of it.
  - bundled dependencies: Regardless of source, this is a version number that is purely for informational purposes.
  - registry sources: This is a version number. (eg, `1.2.3`)
  - git sources: This is a git specifier with resolved committish. (eg, `git+https://example.com/foo/bar#115311855adb0789a0466714ed48a1499ffea97e`)
  - http tarball sources: This is the URL of the tarball. (eg, `https://example.com/example-1.3.0.tgz`)
  - local tarball sources: This is the file URL of the tarball. (eg `file:///opt/storage/example-1.3.0.tgz`)
  - local link sources: This is the file URL of the link. (eg `file:libs/our-module`)
- integrity: A `sha512` or `sha1` Standard Subresource Integrity string for the artifact that was unpacked in this location. For git dependencies, this is the commit sha.
- resolved: For registry sources this is path of the tarball relative to the registry URL. If the tarball URL isn't on the same server as the registry URL then this is a complete URL. `registry.npmjs.org` is a magic value meaning "the currently configured registry".
- bundled: If true, this is the bundled dependency and will be installed by the parent module. When installing, this module will be extracted from

the parent module during the extract phase, not installed as a separate dependency.

- dev: If true then this dependency is either a development dependency ONLY of the top level module or a transitive dependency of one. This is false for dependencies that are both a development dependency of the top level and a transitive dependency of a non-development dependency of the top level.
- optional: If true then this dependency is either an optional dependency ONLY of the top level module or a transitive dependency of one. This is false for dependencies that are both an optional dependency of the top level and a transitive dependency of a non-optional dependency of the top level.
- requires: This is a mapping of module name to version. This is a list of everything this module requires, regardless of where it will be installed. The version should match via normal matching rules a dependency either in our `dependencies` or in a level higher than us.
- dependencies: The dependencies of this dependency, exactly as at the top level.

See also

- npm shrinkwrap
- npm-shrinkwrap.json
- package.json
- npm install