libnpmversion

Library to do the things that 'npm version' does.

USAGE

```
const npmVersion = require('libnpmversion')

// argument can be one of:
// - any semver version string (set to that exact version)
// - 'major', 'minor', 'patch', 'pre{major,minor,patch}' (increment at
//   that value)
// - 'from-git' (set to the latest semver-lookin git tag - this skips
//   gitTagVersion, but will still sign if asked)
npmVersion(arg, {
  path: '/path/to/my/pkg', // defaults to cwd

  allowSameVersion: false, // allow tagging/etc to the current version
  preid: '', // when arg=='pre', define the prerelease string, like 'beta' etc.
  tagVersionPrefix: 'v', // tag as 'v1.2.3' when versioning to 1.2.3
  commitHooks: true, // default true, run git commit hooks, default true
  gitTagVersion: true, // default true, tag the version
  signGitCommit: false, // default false, gpg sign the git commit
  signGitTag: false, // default false, gpg sign the git tag
  force: false, // push forward recklessly if any problems happen
  ignoreScripts: false, // do not run pre/post/version lifecycle scripts
  scriptShell: '/bin/bash', // shell to run lifecycle scripts in
  message: 'v%s', // message for tag and commit, replace %s with the version
  silent: false, // passed to @npmcli/run-script to control whether it logs
}).then(newVersion => {
  console.error('version updated!', newVersion)
})
```

Description

Run this in a package directory to bump the version and write the new data back to `package.json`, `package-lock.json`, and, if present, `npm-shrinkwrap.json`.

The `newversion` argument should be a valid semver string, a valid second argument to semver.inc (one of `patch`, `minor`, `major`, `prepatch`, `preminor`, `premajor`, `prerelease`), or `from-git`. In the second case, the existing version will be incremented by 1 in the specified field. `from-git` will try to read the latest git tag, and use that as the new npm version.

If run in a git repo, it will also create a version commit and tag. This behavior is controlled by `gitTagVersion` (see below), and can be disabled by setting `gitTagVersion: false` in the options. It will fail if the working directory is not clean, unless `force: true` is set.

If supplied with a `message` string option, it will use it as a commit message when creating a version commit. If the `message` option contains `%s` then that will be replaced with the resulting version number.

If the `signGitTag` option is set, then the tag will be signed using the `-s` flag to git. Note that you must have a default GPG key set up in your git config for this to work properly.

If `preversion`, `version`, or `postversion` are in the `scripts` property of the package.json, they will be executed in the appropriate sequence.

The exact order of execution is as follows:

1. Check to make sure the git working directory is clean before we get started. Your scripts may add files to the commit in future steps. This step is skipped if the `force` flag is set.
2. Run the `preversion` script. These scripts have access to the old `version` in package.json. A typical use would be running your full test suite before deploying. Any files you want added to the commit should be explicitly added using `git add`.
3. Bump `version` in `package.json` as requested (`patch`, `minor`, `major`, explicit version number, etc).
4. Run the `version` script. These scripts have access to the new `version` in package.json (so they can incorporate it into file headers in generated files for example). Again, scripts should explicitly add generated files to the commit using `git add`.
5. Commit and tag.
6. Run the `postversion` script. Use it to clean up the file system or automatically push the commit and/or tag.

Take the following example:

```
{
  "scripts": {
    "preversion": "npm test",
    "version": "npm run build && git add -A dist",
    "postversion": "git push && git push --tags && rm -rf build/temp"
  }
}
```

This runs all your tests, and proceeds only if they pass. Then runs your `build` script, and adds everything in the `dist` directory to the commit. After the commit, it pushes the new commit and tag up to the server, and deletes the `build/temp` directory.

API

`npmVersion(newversion, options = {}) -> Promise<String>`

Do the things. Returns a promise that resolves to the new version if all is well, or rejects if any errors are encountered.

Options

`path` **String**

The path to the package being versionified. Defaults to process.cwd().

`allowSameVersion` **Boolean**

Allow setting the version to the current version in package.json. Default `false`.

`preid` String

When the `newversion` is pre, premajor, preminor, or prepatch, this defines the prerelease string, like 'beta' etc.

`tagVersionPrefix` String

The prefix to add to the raw semver string for the tag name. Defaults to `'v'`. (So, by default it tags as 'v1.2.3' when versioning to 1.2.3.)

`commitHooks` Boolean

Run git commit hooks. Default true.

`gitTagVersion` Boolean

Tag the version, default true.

`signGitCommit` Boolean

GPG sign the git commit. Default `false`.

`signGitTag` Boolean

GPG sign the git tag. Default `false`.

`force` Boolean

Push forward recklessly if any problems happen. Default `false`.

`ignoreScripts` Boolean

Do not run pre/post/version lifecycle scripts. Default `false`.

`scriptShell` String

Path to the shell, which should execute the lifecycle scripts. Defaults to `/bin/sh` on unix, or `cmd.exe` on windows.

`message` String

The message for the git commit and annotated git tag that are created.