Whirlwind tour of (correct) npm usage

This is a quick tour of how to get started with NPM, how to use it, and how to fix it.

I'm available for tutoring and code review :)

Starting a new project

Create a folder for your project, preferably a Git repository. Navigate into that folder, and run:

```
npm init
```

It will ask you a few questions. Hit `Enter` without input if you're not sure about a question, and it will use the default.

You now have a `package.json`.

If you're using Express: Please don't use `express-generator`. It sucks. Just use `npm init` like explained above, and follow the 'Getting Started' and 'Guide' sections on the Express website. They will teach you all you need to know when starting from scratch.

Installing a package

All packages in NPM are *local* - that is, specific to the project you install it in, and actually installed *within* that project. They are also nested - if you use the `foo` module, and `foo` uses the `bar` module, then you will have a `./node_modules/foo/node_modules/bar`. This means you pretty much never have version conflicts, and can install as many modules as you want without running into issues.

All modern versions of NPM will 'deduplicate' and 'flatten' your module folder as much as possible to save disk space, but as a developer you don't have to care about this - it will still work like it's a tree of nested modules, and you can still assume that there will be no version conflicts.

You install a package like this:

```
npm install packagename
```

While the packages themselves are installed in the `node_modules` directory (as that's where the Node.js runtime will look for them), that's only a temporary install location. The *primary* place where your dependencies are defined, should be in your `package.json` file - so that they can be safely updated and reinstalled later, even if your `node_modules` gets lost or corrupted somehow.

In older versions of npm, you had to manually specify the `--save` flag to make sure that the package is saved in your `package.json`; that's why you may come across this in older articles. However, modern versions of NPM do this automatically, so the command above should be enough.

One case where you *do* still need to use a flag, is when you're installing a module that you just need for developing your project, but that isn't needed when actually *using* or *deploying* your project. Then you can use the `--save-dev` flag, like so:

```
npm install --save-dev packagename
```

Works pretty much the same, but saves it as a development dependency. This allows a user to install just the 'real' dependencies, to save space and bandwidth, if they just want to use your thing and not modify it.

To install everything that is declared in `package.json`, you just run it without arguments:

```
npm install
```

When you're using Git or another version control system, you should add `node_modules` to your ignore file (eg. `.gitignore` for Git); this is because *installed* copies of modules may need to be different depending on the system. You can then use the above command to make sure that all the dependencies are correctly installed, after cloning your repository to a new system.

Semantic versioning

Packages in NPM usually use semantic versioning; that is, the changes in a version number indicate what has changed, and whether the change is breaking. Let's take **1.2.3** as an example version. The components of that version number would be:

- **Major version number:** 1
- **Minor version number:** 2

- **Patch version number:** 3

Depending on which number changes, there's a different kind of change to the module:

- **Patch version upgrade (eg. `1.2.3` -> `1.2.4`):** An internal change was made, but the API hasn't changed. It's safe to upgrade.
- **Minor version upgrade (eg. `1.2.3` -> `1.3.0`):** The API has changed, but in a backwards-compatible manner - for example, a new feature or option was added. It's safe to upgrade. You may still want to read the changelog, in case there's new features that you want to use, or that you were waiting for.
- **Major version upgrade (eg. `1.2.3` -> `2.0.0`):** The API has changed, and is not backwards-compatible. For example, a feature was removed, a default was changed, and so on. It is **not** safe to upgrade. You first need to read the changelog, to see whether the changes affect your application.

Most NPM packages follow this, and it gives you a lot of certainty in what upgrades are safe to carry out, and what upgrades aren't. NPM explicitly adopts semver in its package.json as well, by introducing a few special version formats:

- `~1.2.3`: Allow automatic patch upgrades, but not minor or major upgrades. Upgrading to `1.2.4` is allowed, but upgrading to `1.3.0` or `2.0.0` is not. You still can't downgrade below `1.2.3` - for example, `1.2.2` is *not* allowed.
- `^1.2.3`: Allow automatic patch and minor upgrades, but not major upgrades. Upgrading to `1.2.4` or `1.3.0` is allowed, but upgrading to `2.0.0` is not. You still can't downgrade below `1.2.3` - for example, `1.2.2` or `1.1.0` are *not* allowed.
- `1.2.3`: Require this specific version. No upgrades are allowed. You will rarely need this - only for misbehaving packages, really.
- `*`: Allow upgrades to whatever the latest version is. You should **never** use this.

By default, NPM will automatically use the **^1.2.3** notation, which is usually what you want. Only configure it otherwise if you have an explicit reason to do so.

A special case are `0.x.x` versions - these are considered to be 'unstable', and the rules are slightly different: the *minor* version number indicates a breaking change, rather than the major version number. That means that `^0.1.2` will allow an upgrade to `0.1.3`, but *not* to `0.2.0`. This is commonly used for pre-release testing versions, where things may wildly change with every release.

If you end up publishing a module yourself (and you most likely eventually will), then definitely adhere to these guidelines as well. They make it a lot easier for developers to keep dependencies up to date, leading to considerably less bugs and security issues.

Global modules

Sometimes, you want to install a command-line utility such as `peerflix`, but it doesn't belong to any particular project. For this, there's the `--global` or `-g` flag:

```
npm install -g peerflix
```

If you used packages from your distribution to install Node, you may have to use `sudo` for global modules.

Never, ever, ever use global modules for project dependencies, ever. It may seem 'nice' and 'efficient', but you will land in dependency hell. It is not possible to enforce semver constraints on global modules, and things will spontaneously break. All the time. Don't do it. Global modules are *only for project-independent, system-wide, command-line tools*.

This applies even to development tools for your project. Different projects will often need different, incompatible versions of development tools - so those tools should be installed *without* the global flag. For local packages, the binaries are all collected in `node_modules/.bin`. You can then run the tools like so:

```
./node_modules/.bin/eslint
```

NPM is broken, and I don't understand the error!

The errors that NPM shows are usually not very clear. I've written a tool that will analyze your error, and try to explain it in plain English. It can be found here.

My dependencies are broken!

If you've just updated your Node version, then you may have native (compiled) modules that were built against the old Node version, and that won't work with the new one. Run this to rebuild them:

```
npm rebuild
```

My dependencies are still broken!

Make sure that all your dependencies are declared in `package.json`. Then just remove and recreate your `node_modules`:

```
rm -rf node_modules
npm install
```

Revision #1
Created 11 December 2024 18:14:33 by joepie91