Short intro to npm dependencies

Installing dependencies

We have dependencies - some constraints that tell what packages and what versions must be present.

Sometimes there are also implicit constraints of the ecosystem. For example, in python world all packages are on the same level, there can only be a single version of each package in the environment.

We can view the "installation" process as two major steps:

1. Find a solution to dependency constraints – a set of packages (= name + version) that satisfies them
2. Download that set of packages and install them

A simple approach

Just a list of dependencies.

The app has a `package.json` file with `dependencies` section.

```
{
  "name": "frontend",
  "dependencies": {
    "vue": "^2.6.11", // ⩾2.0.0 <3.0.0
    "vuex": "^2.3.1", // ⩾2.0.0 <3.0.0
  },
}
```

When we want to install packages we run `npm install`

NPM finds a solution and installs them

Issues of that approach

Every time we’re trying to find a new solution to dependency constraints.

If the algorithm is non-deterministic, every new installation may result in different set of packages.

Even if the algorithm is deterministic, it can still result in a different set of packages if a set of (remotely) available packages changes. For example, a new version came out.

Solving the issue: Option 1

Always specify the exact version for all packages

```
  {
    "name": "frontend",
    "dependencies": {
-     "vue": "^2.6.11", // ⩾2.0.0 <3.0.0
-     "vuex": "^2.3.1", // ⩾2.0.0 <3.0.0
+     "vue": "2.6.11", // ==2.6.11
+     "vuex": "2.3.1", // ==2.3.1
    },
  }
```

This works!

We lost an option of "one-click" upgrading everything, but one can argue it’s rarely needed.

Solving the issue: Option 2

Lockfiles

After finding a solution that satisfies our dependencies, we save it to a file

Then we can use it multiple times and on different machines to guarantee that teammates, deployments, and continuous integration install exactly the same dependencies

We don’t spend time on finding a solution on future installations

Lockfiles in NPM

Lockfiles are the current default approach

`npm install`

1. Finds a solution to dependency constraints
2. Writes it to `package-lock.json`
3. Installs packages

- it does not honor `package-lock.json` contents, will always overwrite it with the new tree of packages
- the only guarantee it gives is that after it's finished, the installed set of packages will satisfy package.json dependencies

`npm ci`

1. Cleans already installed packages
2. Installs all packages specified in `package-lock.json`

- does not ever write to `package.json` or `package-lock.json`

Cheatsheet

Commands to use for popular cases

Do not manually change `package-lock.json`
Do not manually change `package.json`'s `dependencies`
Commit `package.json` and `package-lock.json` to the repo

Install packages in any non-local environment: `npm ci`
Install packages in a clean local environment: `npm ci`
Install new set of packages locally if they were changed (like after a pull or checkout): `npm ci`

Reset installed packages locally (e.g. you changed dependencies and decided to not proceed): discard local changes to `package.json` and `package-lock.json` with git, then `npm ci`

Add a new dependency: `npm install --save[-dev] vue@^2.6`

Change version requirements of a dependency: `npm install --save[-dev] vue@^2.6`

Remove a dependency: `npm uninstall --save[-dev] vue`