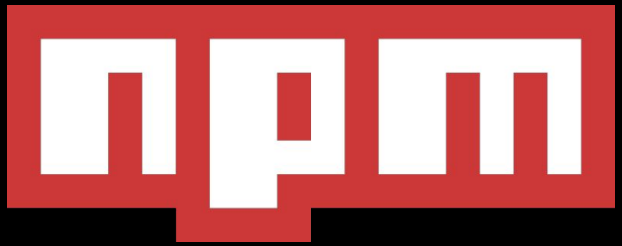
Full Stack III - Session 1.2

Introduction to NPM



Topics

- NPM and NPM Registry
- NPM CLI Commands
- Semantic Versioning

NPM and NPM Registry

NPM

- NPM stands for Node Package Manager
- It is the default package manager for the JavaScript runtime environment Node.js
- You can install, share and manage node.js packages.
- npm consists of three components:
  - Website
  - Registry
  - CLI

NPM cont..

Website:

npm official website is https://www.npmjs.com/. Using this website you can find packages, view documentation, share and publish packages.

Registry:

npm registry is a large database consisting of more than half a million packages. Developers download packages from the npm registry and publish their packages to the registry.

Use npm to...

- Download standalone tools and packages you can use right away.
- Share code with any npm user, anywhere.
- Update applications easily when underlying code needs updating.
- Manage multiple version of code and code dependencies.

NPM CLI Commands

NPM - CLI (Command Line Interface) common commands

- **npm install** - installs a package
- **npm uninstall** - uninstalls a package
- **npm init** - creates a package.json file
- **npm test** - tests a package
- **npm ls** - lists installed packages
- **npm help** - get help on npm

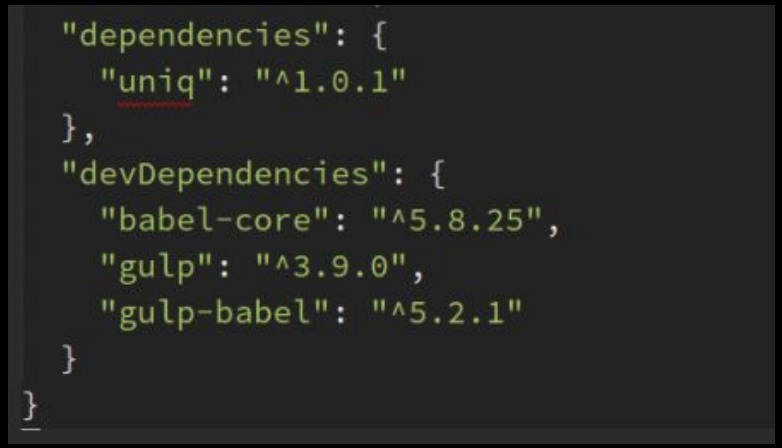
- Full list of commands can be found at https://docs.npmjs.com/cli/

NPM - init

- Npm init - will create a package.json file
- * version - a version number that should be understandable by node-semver.

NPM - CLI (Command Line Interface) flags

- **npm install --global** - installs global only to your local machine ie. express
- **npm install --save** - installs as dependency to your package.json file ie.
- **npm install --save--dev** - installs as a dev dependency to your package.json file
    - This allows anyone who might develop on or use the project to install the dependencies with a simple npm instal command.



Open Source Community

Video - NPM

Leftpad - “kik” incident

```
module.exports = leftpad;
function leftpad (str, len, ch) {
  str = String(str);
  var i = -1;
  if (!ch && ch !== 0) ch = ' ';
  len = len - str.length;
  while (++i < len) {
    str = ch + str;
  }
  return str;
}
```

- Azer Koculu created a npm package callled "kik", a CLI for "kick-starting" projects. At the same time a corporation Stratton had a patent for Kik.
- There was a legal dispute, Koculu stood his ground and npm sided with the company. The developer deleted his package, plus the other 273 modules he had registered in npm
- One of the modules, "leftpad" is 17 lines of code used to right-justify text. It started to break very large packages including Babel, the JavaScript compiler and React
- First time in history npm had to restore an unpublished npm package

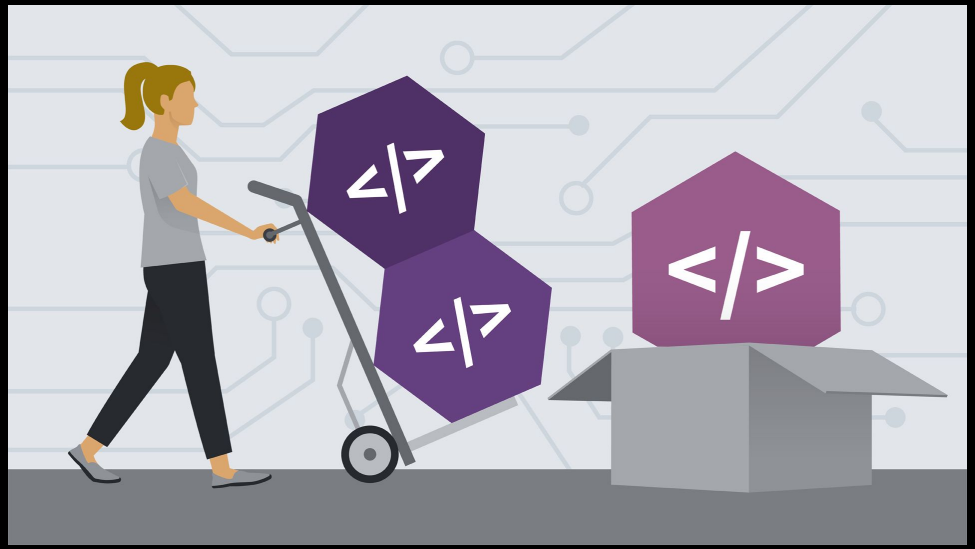
Package.json

Package.json

The package.json file is core to the Node.js ecosystem and is a basic part of understanding and working with Node.js, npm, and even modern JavaScript. The package.json is used as what equates to a manifest about applications, modules, packages, and more - it's a tool to that's used to make modern development streamlined, modular, and efficient.

As a developer in the Node.js ecosystem, understanding the basics of package.json is one of the first steps to really kicking off your development experience with Node.js.

Package.json



Because of how **essential** understanding the basics of package.json is to development with Node.js, I've gone through and outlined some of the most common and important properties of a package.json file that you'll need to use package.json effectively.

“Name” property

The name property in a package.json file is one of the fundamental components of the package.json structure. At its core, name is a string that is *exactly* what you would expect - the name of the module that the package.json is describing.

Inside your package.json, the name property as a string would look something like this:

```
"name": "metaverse"
```

“version” property

The version property is a key part of a package.json, as it denotes the current version of the module that the package.json file is describing.

Inside your package.json, the version property as a string using semver could look like this:

```
"version": "5.12.4"
```

“description” property

The description property of a package.json file is a string that contains a human-readable description about the module - basically, it's the module developer's chance to quickly let users know what *exactly* a module does. The description property is frequently indexed by search tools like npm search and the npm CLI search tool to help find relevant packages based on a search query.

Inside your package.json, the description property would look like this:

```
   "description": "The Metaverse virtual reality. The final outcome of all virtual worlds, augmented reality, and the Internet."
```

“keywords” property

```
"keywords": [
    "metaverse",
    "virtual reality",
    "augmented reality",
    "snow crash"
]
```

The keywords property inside a package.json file is, as you may have guessed, a collection of keywords about a module. Keywords can help identify a package, related modules and software, and concepts.

The keywords property is always going to be an array, with one or more strings as the array's values - each one of these strings will, in turn, be one of the project's keywords.

“main” property

The main property of a package.json is a direction to the entry point to the module that the package.json is describing. In a Node.js application, when the module is called via a require statement, the module's exports from the file named in the main property will be what's returned to the Node.js application.

```
"main": "app.js",
```

“repository” property

The repository property of a package.json is an array that defines *where* the source code for the module lives. Typically, for open source projects, this would be a public GitHub repo, with the repository array noting that the type of version control is git, and the URL of the repo itself. One thing to note about this is that it's not just a URL the repo can be accessed from, but the full URL that the *version control* can be accessed from.

```
"repository": {
    "type": "git",
    "url": "https://github.com/bnb/metaverse.git"
}
```

“scripts” property

```
"scripts": {
    "build": "node app.js",
    "test": "standard"
}
```

The scripts property of a package.json file is simple conceptually, but is complex functionally to the point that it's used as a build tool by many.

At its simplest, the scripts property takes an object with as many key/value pairs as desired. Each one of the keys in these key/value pairs is the name of a command that can be run. The corresponding value of each key is the actual command that *is* run. Scripts are frequently used for testing, building, and streamlining of the needed commands to work with a module.

“dependencies” property

```
"dependencies": {
  "async": "^0.2.10",
  "npm2es": "~0.4.2",
  "optimist": "~0.6.0",
  "request": "~2.30.0",
  "skateboard": "^1.5.1",
  "split": "^0.3.0",
  "weld": "^0.2.2"
},
```

The dependencies property of a module's package.json is where dependencies - the *other* modules that *this* module uses - are defined.

The dependencies property takes an object that has the name and version at which each dependency should be used. Tying things back to the version property defined earlier, the version that a module needs is defined. Do note that you'll frequently find carets (^) and tildes (~) included with package versions.

“devdependencies” property

The devDependencies property of a package.json is almost identical to the dependencies property in terms of structure, with a key difference.

The dependencies property is used to define the dependencies that a module needs to run in *production*.

The devDependencies property is *usually* used to define the dependencies the module needs to run in *development*.

```
"devDependencies": {
    "escape-html": "^1.0.3",
    "lucene-query-parser": "^1.0.1"
}
```

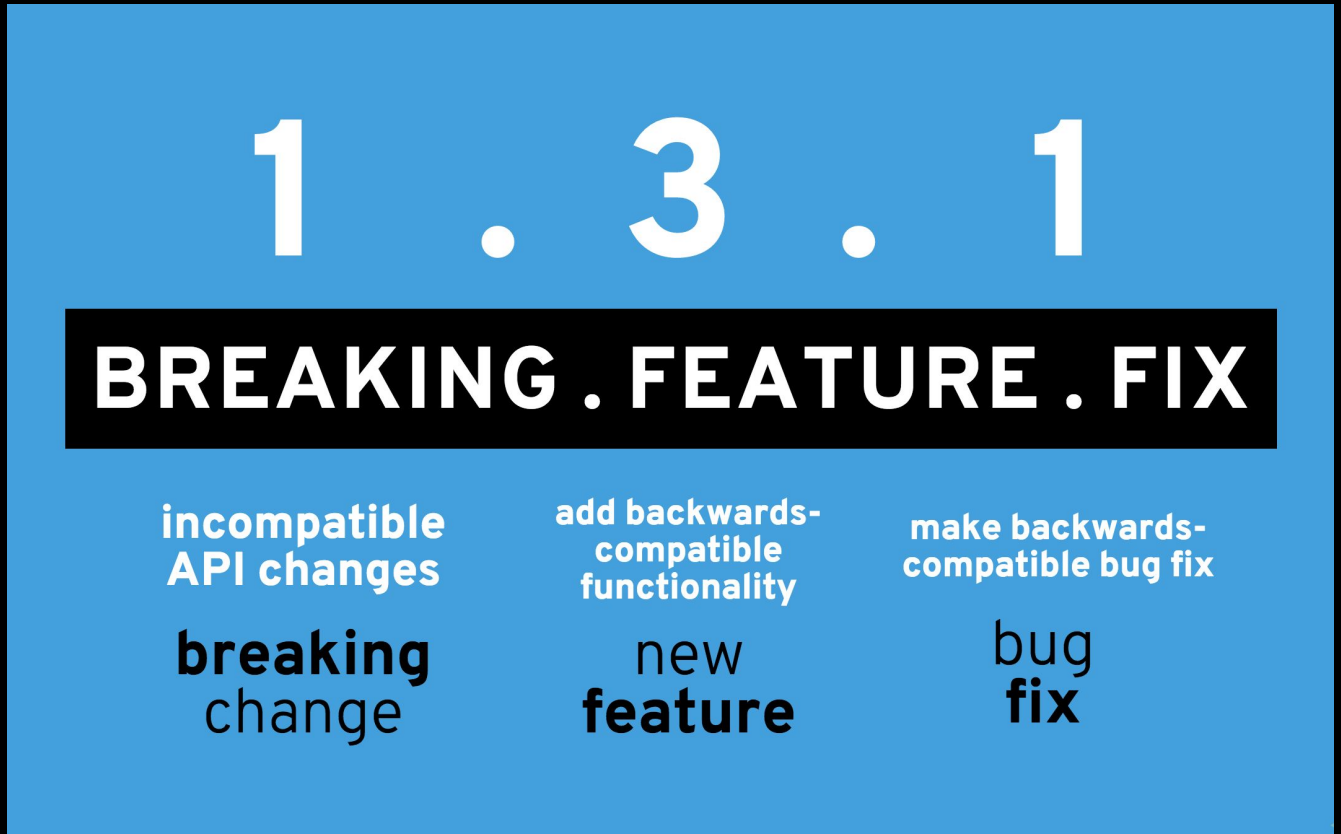
Video - package.json

Semantic Versioning

NPM - Semver

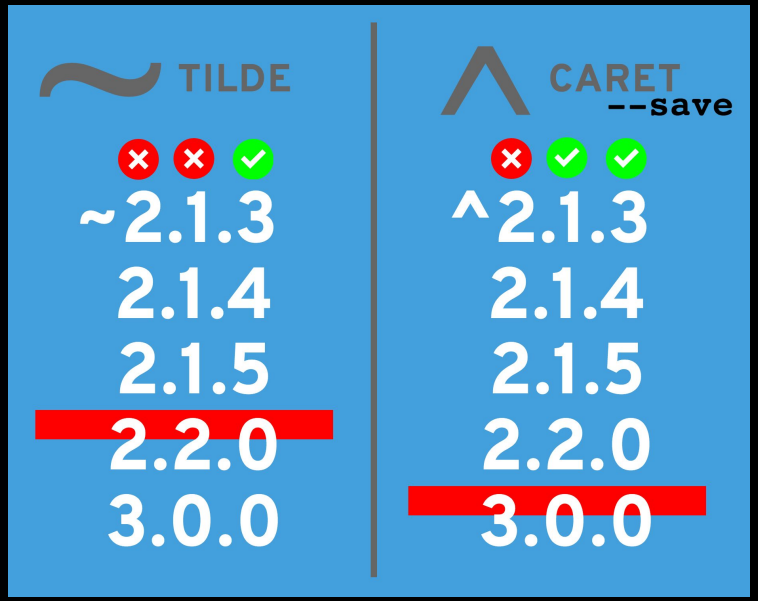
- Semver - stands for Semantic Versioning https://semver.org/
- Semantic versioning is a way to specify package versions with a three part version number
- Logical comparison operators, and some wildcarding allow you to control which version(s) of a package you want

Semantic Versioning



Semantic Versioning

Semver - Range Specifiers



Symbol	Dependency	Versions	Changes
caret (^)	^3.9.2	3.*.*	– backwards compatible new functionality – old functionality deprecated, but operational – large internal refactor – bug fix
tilde (~)	~3.9.2	3.9.*	– bug fix

Video - Installing Nodemon