Third-Party Modules

SD540 Server-Side Programming

Maharishi University of Management

Masters of Software Development

Associate Professor Asaad Saad

Maharishi International University - Fairfield, Iowa

Modules in NodeJS

Node **core modules** *(shipped with Node)*

User-defined **modules** *(files created by user)*

3rd-party code modules *(`npm i moduleName`, **`node_modules`** folder)*

3rd-party **development modules** *(`npm i moduleName -D`, **`node_modules`** folder)*

3rd-party **global modules** *(`npm i moduleName -g`, global OS folder, we cannot import)*

User Defined Modules

Imports are **read-only** live bindings to the original exported variable in the module.

- "read-only" means you can't directly modify them.
- "live" means that you can see any indirect modifications made to them.

Changes must be done in **immutable** way.

```
export let data = [];

export function addItem(item) {
    data = [...data, item]
}
```

Packages and Package Manager

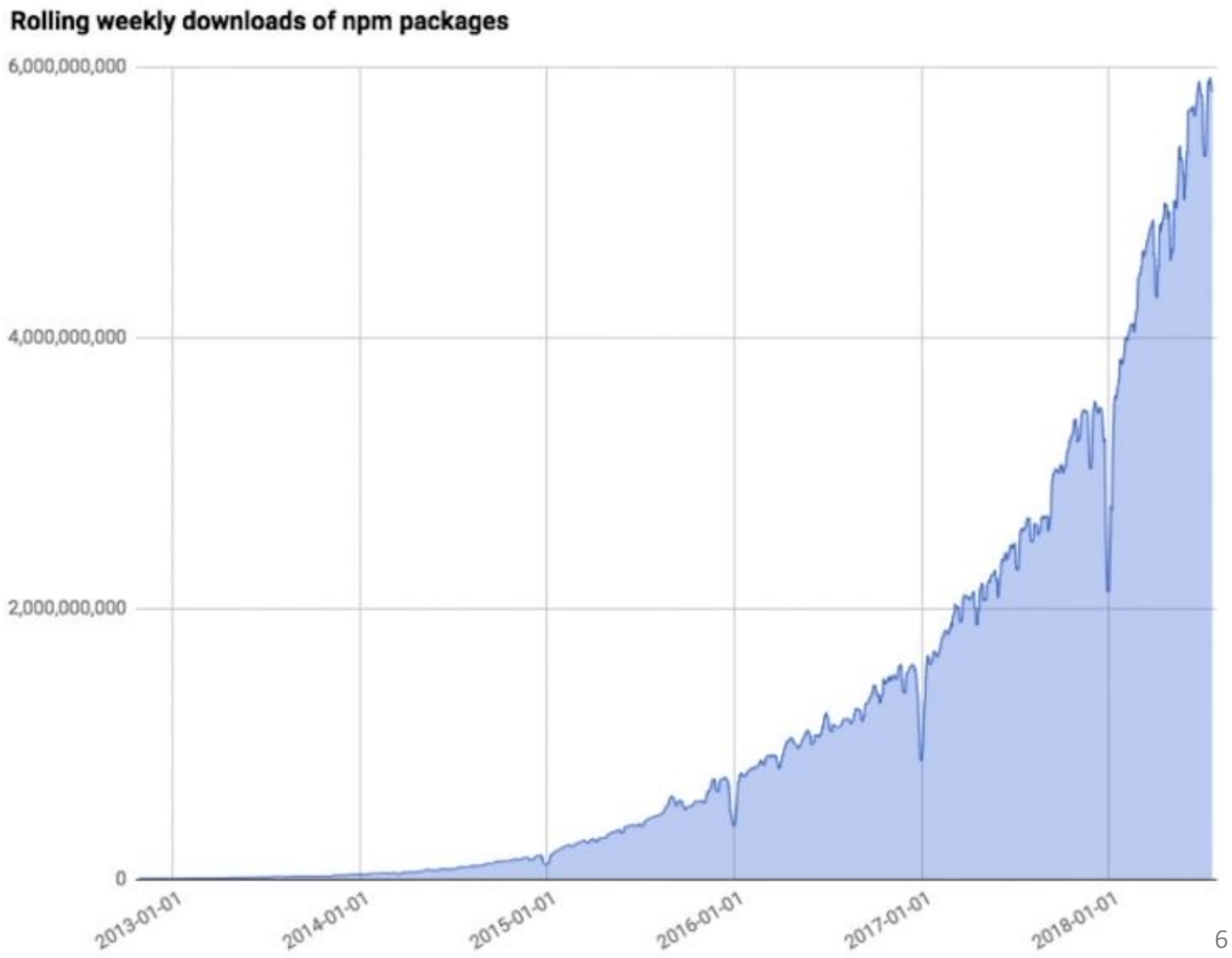
Package is a collection of code (module) you can use it in your code. It's managed by package manager.

Dependencies: code (module) that another set of code (module) depends on to work. If you use that code in your app, it is a dependency and your app depends on it. And that code you depend on might depend on another package. this is why we need a strong package manager to manage all dependencies and version control.

Package management system: software that automates installing and updating packages. Deals with what version you have or need, and manages dependencies. Update them when needed.

NPM is the largest software registry in the world

https://www.npmjs.com



Laurie Voss

npm: Node Package Manager

The npm registry: source of all codes we are installing: www.npmjs.com

npm is not part of node but a dependency complementing Node.

Another package manager is Yarn from Facebook www.yarnpkg.com

When we install a package:

- Notice dependencies changes in **`package.json`**
- notice folder: **`node_modules`**
- This structure separates code from dependencies, at any later time we can copy our code and run: **`npm install`** (will read all dependencies and install them for us) helpful when we deploy.
- We have three kind of packages: dependency, development and global.
- Add **`.gitignore`** to your project listing **`node_modules`** folder

npm Demo

```
npm –v          // will print npm version
npm init        // will create package.json
npm i <package> // install & audit from last commit of git repo
                   npm will update package.json automatically
npm i <package> -D // install a development package
npm i <package> -g  // install a global package

npm update      // check versions in package.json and update

npm prune       // delete packages that are not defined in package.json

npx <package>   // install a global package, run, then delete

npm home <package> // open browser to package homepage
npm repo <package> // open browser to package repository
```

Code Dependencies

Check if they include types, if not, install the definitely typed package separately.

Check Weekly Downloads, a good package has high amount of downloads.

Check Git issues, a good package is well maintained and should not have a lot of opened issues.

Consult with your manager before you install any package.

Semantic Versioning

We will use semantic versioning to Giving a version of code meaning:

MAJOR.MINOR.PATCH

- **PATCH**: Some bugs were fixed. Your code will work fine
- **MINOR**: some new features were added. Your code will work fine.
- **MAJOR**: Big changes. Your code will break (maybe)

www.semver.org

package.json Manifest

```
{
        "name": "nodejs-test-application",
        "version": "1.0.0",
        "description": "NodeJS Test App",
        "main": "app.js",
        "scripts": {
                "scriptName": "command"   // npm run scriptName
        },
        "author": "Asaad Saad",
        "license": "ISC",
        "dependencies": {
                "moment": "^2.10.6" // MAJOR.MINOR.PATCH
        }                          // ^ (caret) only update minor and patches
}                                  // ~ (tilde) only update patches
```

Development Dependencies

Needed only while developing the app

Declared in **`package.json`** as **`devDependencies`**

They live locally in **`node_modules`**

You cannot import and use them from your code

Example: `npm i nodemon –D`

Global Dependencies

These are binary executables that assign a new global command

They are not declared in **`package.json`**

They are not found in **`node_modules`**

You cannot import and use them from your code.

Example:
```
npm i http-server -g
http-server -o
```

npx

When installing a global package globally, it becomes outdated.

It is common to use **npx** instead, which installs, runs, and deletes the global package when it finishes.

Example: `npx http-server -o`

`package-lock.json`

- The purpose of the **`package-lock`** is to avoid the situation where installing modules from the same **`package.json`** results in two different installs.
- **`package-lock`** is a large list of each dependency listed in your **`package.json`**, the specific version that should be installed, the location of the module, a hash that verifies the integrity of the module, and a list of its dependencies, so the install it creates will be the same, every single time.

How is it used?

- It is automatically generated the first time and every time you install or update your dependencies. Do not modify **`package-lock`** manually.
- When **`package-lock`** is found, it is used to download the exact version of dependencies.
- **`package.json`** overrules the **`package-lock`** if **`package.json`** has been updated.
- You should commit your **`package-lock`** to source control
- Do not commit **`node_modules`** to source control