**What is NodeJS**

* Node.js® is a JavaScript platform built on Chrome’s V8 JS engine.
* Uses an event-driven, non-blocking I/O model that makes it lightweight and efficient.
* Is used to develop I/O intensive web applications
* Open source
* Users include Microsoft, LinkedIn, IBM, Paypal, Netflix, Ebay,

even NASA.

* Available to all major operating systems.

**Installation**

* Found on <https://nodejs.org>.
* Latest versions come in 2 flavours: **LTS** and **Current.**
* Both are stable!
* The Current version is the latest and may change rapidly, which may be inconvenient for consistent developing environment.
* LTS (**Long Term Support**) version is recommended as it is updated and maintained for longer and predefined schedules.

This requires less frequent updating , thus is more “stable”

* DL and easily install.

In the console enter **node --version** to verify node is installed

* Multiple Node versions may be installed **and even run** simultaneously using **NVM**

**Testing the installation: Hello World App**

1. Open a console/terminal
2. Type **node --version** to verify node is installed.
3. Create **HelloWorld.js** (e.g. in windows: NUL > helloWorld.js)
4. In the created file type: console.log(‘Hello World’);
5. In the console type: **node helloworld.js**

**Debugging in Node.JS - quick walkthrough**

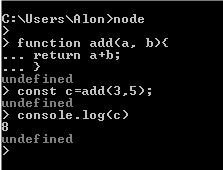
The node recommended tool - **--inspect**

1. Type node --inspect

**REPL - Read Eval Print Loop**

Bundled inside Node, REPL represents a console-like interactive environment for experimenting with Node code and debugging JS code.

1. In the console just type **node**
2. Enter some JS code:



Read - reads user input, parses it into JS data structure and store in memory

Eval - evaluate the data structure

Print - output the results

Loop - run this command until dbl entering ctrl-c

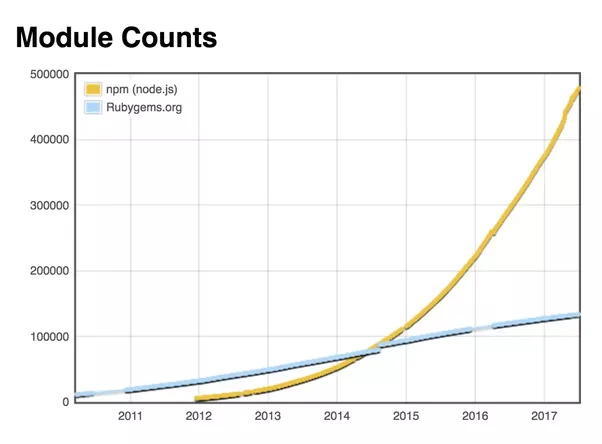
**A note about Node modules and packages**

Module - Is a library. In Node, files and modules are in one-to-one relationship, e.g.

foo.js loads the module circle.js in the same directory

Package - Is a one or more modules packaged together.

**Package Management**

* The intense growth of JS users and modules called for a reliable packaging mechanism.
* The recursion of dependencies may lead to multiple versions of the same dependency, which makes the process a challenge:

Consider this scenario:

MyApp ⇒ PackageA ⇒ PackageB V0.5

MyApp ⇒ PackageC ⇒ PackageB V1.1

What version of PackageB will be installed in MyApp’s packages?

And things could even get worse…

* Node.JS package managers make it easy for JavaScript developers to share and reuse code, and to update the code that you're sharing.
* Node packages all have a description file:

Package.json

**Package.json basic example:**

{

"name": "multo-grid",

"version": "5.12.4”,

“license": "MIT",

"description": "The best grid ever”,

"keywords": [  
 "grid",  
 "virtualisation",  
 ],

"main": "app.js",

"repository": {  
 "type": "git",  
 "url": "https://github.com/trainologic-mock/multo-grid.git"  
 },

"scripts": {

"build": "node app.js",  
 "test": "standard"  
 },

"dependencies": {  
 "async": "^0.2.10",  
 "npm2es": "~0.4.2",

"<http://asdf.com/asdf.tar.gz>"  
 },

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

}

**Some Important Properties**

These are the most important and are required:

* Name - The name that will be used when installing this package.

There are some rules.

* Version - The current version number

Together they form the package identifier

Some more typical properties:

* Description - More than the name implies, this is frequently indexed

for finding the package inside npm

* Keywords - a string array , also for searching purposes
* main - Entry point that will export and will be used to import the

module

* Repository - Specifies where the code lives. Useful for contributors.
* Scripts - Key/values of actions(values) to run at various stages (keys) in the module’s lifecycle
* Dependencies - Mappings of package names to a version range or a URL.

DevDependencies - Same as above, but are not needed for users installing the

package. Includes things like transpilers and test harnesses

**NPM**

There are several package managers:

Bower, Yarn, npm



npm is the best known and is automatically installed with Node.

* You can update to the latest npm version by typing:

npm install npm@latest -g

**Common npm commands**

Install/Uninstall - Install/uninstall a package. comes in 2 location flavors -

local - to be imported into your project using an import statement

global - to be used as a command line tool, e.g. jest.

Examples:

$ npm install react --save

Install react locally and update the dependencies in package.json

Installed packages will appear in the location’s node\_modules folder

Note: As of npm ver 5+ --save is the default mode and can be neglected.

$ npm install jest -g

Install jest globally. Package will be saved in node default node\_module folder.

Installed packages will appear in the location’s node\_modules folder

update - update all listed packages to their latest versions, and install missing ones, using the local/global *paradigm.*

*Updates everything in the specified location if no list is specified.*

$ npm update react , $ npm up -g

ls - List installed packages in the specified location and their dependencies in a tree-structure.

$ npm ls react

Start - Run a script specified in ‘start’ property of ‘scripts’ object in package.json, or node server.jsif no such property exists.

$ npm start

Dedupe

repo - Will try to open the specified package’s repo page.

$ npm repo react - Will open react repo in github

home - Will try to open the specified package’s home page.

$ npm home react - Will open react home page.

**Node\_modules folder**

* Package dependency tree might get very large, so distributing our fully-installed app may be very slow.
* No need to include installed packages, as their sources are usually in a remote repo anyway.
* Node\_modules is the folder in which packages are installed using npm install. We can neglect this folder when distributing, and the user can simply ‘npm install’ in one go.
* Erasing this folder means you need to re-’npm install’.

**Npm Versioning rules**

* Packages are installed by the version number in their dependencies.
* Most packages use Semver - Semantic Versioning. This version template is of major.minor.patch
* It is important to give npm ‘more space’ in deciding what version of a package to install, i.e. to install the more recent version.
* Packages utilize the semver scheme and special notations to permit npm to just that - set ranges of version numbers that can be installed. The notations come in front of version numbers:

The ^ (caret) states versions upto, but not including, the next major

number may be installed

The ~ (tilde) states versions upto, but not including, the next minor

number may be installed

Also <, >, <=, >= may be used

For example:

Suppose we have in our dependencies:

**"immutable": "^3.8.1"** - version 3.9.6 may be installed, but not

4.0.0.

**"redux": "~3.5.2"** - version 3.5.7 may be installed, but not

3.6.0.

**"lodash": "<3.5.2"** - version must be less than 3.5.2

* Npm will not automatically update newer versions of installed packages.
* Use npm update to do that.

**Lock files**

* Ideally, npm install will act as a pure function - The same package.json will produce the same node\_modules tree.

But in many cases, npm won’t be able to do this:

1. Different versions of npm may have been used to install packages, using slightly using slightly different installation algorithm
2. a new version of a direct semver-range package may have been published since the last time your packages were installed, and thus a newer version will be used.

* To gain control on the current package tree’s version state, npm emits a package-lock.json
* To gai

**Package Resolution at runtime**

Supposehave:

1. In node\_modules - a file called ‘foobar.js’:

function foo() {

return 'bar';

}

module.exports.foo = foo; // (\*)

1. A file called ‘app.js’:

const {foo, bar} = require('foobar'); // (\*\*)

console.log (foo());

When we $ node app.js node does the following:

1. Loads app.js, parses it and evaluates it.
2. It finds the absolute path of foo.js.
3. It loads the foo.js into memory.
4. It wraps foo.js giving it its private scope.
5. It evaluates foo.js :
6. It assigns the return value from exports in (\*) to the requires in (\*\*)
7. It caches foo.js So that when we require this file again, we don’t go over all the steps another time