To install node version manager(nvm)

Download nvm first from browser

Next use windows powershell or cmd…don’t use bash while using nvm

After installation of newer version and everything

Windows PowerShell

Copyright (C) Microsoft Corporation. All rights reserved.

Install the latest PowerShell for new features and improvements! https://aka.ms/PSWindows

PS C:\Users\Admin> nvm --version

1.1.12

PS C:\Users\Admin> nvm list available

| CURRENT | LTS | OLD STABLE | OLD UNSTABLE |

|--------------|--------------|--------------|--------------|

| 22.8.0 | 20.17.0 | 0.12.18 | 0.11.16 |

| 22.7.0 | 20.16.0 | 0.12.17 | 0.11.15 |

| 22.6.0 | 20.15.1 | 0.12.16 | 0.11.14 |

| 22.5.1 | 20.15.0 | 0.12.15 | 0.11.13 |

| 22.5.0 | 20.14.0 | 0.12.14 | 0.11.12 |

| 22.4.1 | 20.13.1 | 0.12.13 | 0.11.11 |

| 22.4.0 | 20.13.0 | 0.12.12 | 0.11.10 |

| 22.3.0 | 20.12.2 | 0.12.11 | 0.11.9 |

| 22.2.0 | 20.12.1 | 0.12.10 | 0.11.8 |

| 22.1.0 | 20.12.0 | 0.12.9 | 0.11.7 |

| 22.0.0 | 20.11.1 | 0.12.8 | 0.11.6 |

| 21.7.3 | 20.11.0 | 0.12.7 | 0.11.5 |

| 21.7.2 | 20.10.0 | 0.12.6 | 0.11.4 |

| 21.7.1 | 20.9.0 | 0.12.5 | 0.11.3 |

| 21.7.0 | 18.20.4 | 0.12.4 | 0.11.2 |

| 21.6.2 | 18.20.3 | 0.12.3 | 0.11.1 |

| 21.6.1 | 18.20.2 | 0.12.2 | 0.11.0 |

| 21.6.0 | 18.20.1 | 0.12.1 | 0.9.12 |

| 21.5.0 | 18.20.0 | 0.12.0 | 0.9.11 |

| 21.4.0 | 18.19.1 | 0.10.48 | 0.9.10 |

This is a partial list. For a complete list, visit https://nodejs.org/en/download/releases

PS C:\Users\Admin> nvm install latest

22.8.0

Downloading node.js version 22.8.0 (64-bit)...

Extracting node and npm...

Complete

npm v10.8.2 installed successfully.

Installation complete. If you want to use this version, type

nvm use 22.8.0

PS C:\Users\Admin> nvm --version

1.1.12

PS C:\Users\Admin> nvm use 22.8.0

Now using node v22.8.0 (64-bit)

PS C:\Users\Admin> nvm ls

\* 22.8.0 (Currently using 64-bit executable)

20.12.2

PS C:\Users\Admin>

U can start using bash console

$ node -v

v20.12.2

Admin@MyLaptop MINGW64 ~

$ node --version

v20.12.2

Admin@MyLaptop MINGW64 ~

$ node -v

v22.8.0

# how to run js files with command using bash

**Admin@MyLaptop MINGW64 ~/Desktop**

**$ mkdir npmv**

**Admin@MyLaptop MINGW64 ~/Desktop**

**$ cd npmv**

**Admin@MyLaptop MINGW64 ~/Desktop/npmv**

**$ touch index.js**

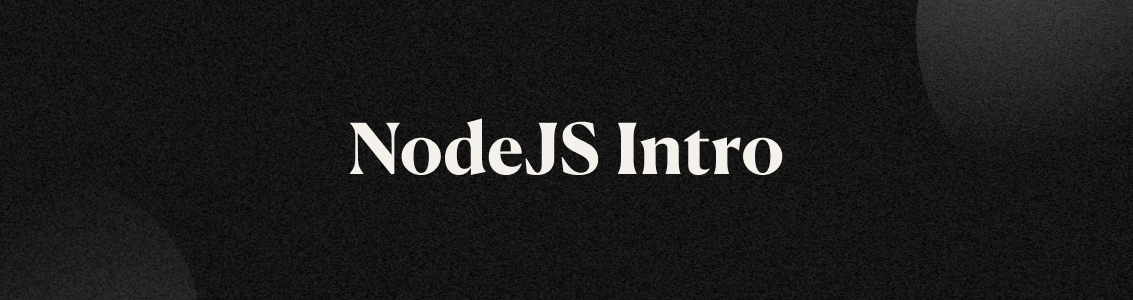
**Admin@MyLaptop MINGW64 ~/Desktop/npmv**

**$ node index.js**

**my first js output**

Lesson 15 of 63

**Day 3 (Part 2) - Node Setup**



**Learning Objectives**

After this lesson, you will know:

* what *runtime* is on a high-level,
* what NodeJS is, a bit of its history, and why you would use it,
* how to use NodeJS on your computer,
* how to use *npm*, which stands for Node Package Manager and
* what is package.json, node\_modules and package-lock.json and why they are a significant part of every Node application.

**What is NodeJS?**

NodeJS (sometimes called just *Node* for short) is a JavaScript **runtime environment** developed by Ryan Dahl that allows us to execute JavaScript code outside of a browser.

**Node has allowed us to run JavaScript anywhere, including web servers** (and robots).

In the past, the only runtime environment for JavaScript was the browser. Chrome, Safari, and Internet Explorer had their JavaScript runtimes, but we *could not* run JavaScript anywhere else. Dahl changed this by extracting the open-source [V8 runtime environment](https://en.wikipedia.org/wiki/Chrome_V8) from Chrome and creating an implementation for servers.

 👀 To summarize, before Node, the only place where you could use JavaScript was a client-side environment - the browsers. Thanks to Node, we can **use JS in the backend** side as well (aka, the server-side), and this makes us, as JavaScript developers, full-stack developers. So thanks, Node. ❤️

Dahl created Node because of the [synchronous limitations of current web frameworks](https://www.infoq.com/interviews/node-ryan-dahl).

**Runtime Environment**

A **runtime environment** is an environment that includes all of the tools and features needed to run a specific program, or in the case of NodeJS, a programming language.

Your computer is dumb. It only understands machine language, 1s, and 0s. Your computer doesn't understand JavaScript, Python, Ruby, Java, and most other languages. These languages have to be translated into [bytecode](https://en.wikipedia.org/wiki/Bytecode), which is then turned into machine code and read by the host operating system, or an interpreter, all while taking into account things like [memory management](https://en.wikipedia.org/wiki/Memory_management), accessing the file system, or using native operating system features.

😲 These terms can sound overwhelming, but take it slow! If you're interested in knowing more about [**how computers process programming languages**](https://www.quora.com/How-do-computers-understand-programming-languages-How-do-you-teach-a-computer-a-language), try to find some answers on Google. Fully understanding these concepts is not the main focus of this course, but we are sure down the road, you will pick it up.

✅ So this is what **NodeJS** does actually: it **translates a JavaScript program into bytecode** for a computers to understand it and to run it.

This topic is enough to teach a class on, and out of scope for this lesson, but in short, JavaScript previously was only able to be run in the browser. Node allows us to run it anywhere!

**What NodeJS is *not***

The following are common misnomers and misconceptions related to Node:

* It **is not a framework** (there are frameworks built on top of NodeJS, though).
* It **is not a programming language** (the language is still JavaScript).

**Why NodeJS?**

Because Node uses JavaScript's *event-driven* and *asynchronous* model, it excels with certain web applications such as chat applications, and apps that need **real-time**, live-updating features.

Another (subjective and debatable) benefit is the ***language of JavaScript being universal***. 🏄

Related to that is the concept of **sharing code between frontend and backend** since both use JavaScript. Sometimes this can be questioned as well since the backend code is very different, even if it is the same language.

Also, Node has a **growing and robust ecosystem**. This includes packages created by the community and continuing open-source development on the runtime itself.

Because Node uses [Google's V8](https://developers.google.com/v8/) behind the scenes, Node also benefits from competition between browsers. As Google improves V8 to compete with [Mozilla's SpiderMonkey](https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey), [Microsoft's Chakra](https://github.com/Microsoft/ChakraCore) and others, so Node reaps the benefits of those improvements as well.

**Our usage of NodeJS**

In this course, we will use Node at first to learn some new features of JavaScript, and write some more complex JavaScript algorithms.

Later on, we will use Node as a web server (we will be talking about servers soon), and we will use a web framework built on top of Node - [ExpressJS](https://expressjs.com/).

**Install** [**NodeJS**](https://nodejs.org/en/)

**Maybe you have Node already?**

*If you finish installing Node in the prework, you might skip this step.* To make sure everything is on its place, run the following in your terminal:

$ node --version # or node -v

and you should get something like this back (this is the latest stabile version in the moment):

v12.6.0

If you get the version below this one and you wish to update, you can:

1. Visit the [NodeJS website](https://nodejs.org/en/)
2. Check the version number of the LTS version
3. Run this command in your terminal (replace 10.16.3 with the latest version number):

$ nvm install 10.16.3

**Installation guide ( *if you don't have Node just yet* )**

Installing Node is relatively easy. We can use brew, apt-get or simply download folder with installations to install Node on Mac, Linux or Windows retrospectively. **But we have to also think about how to keep Node up to date.**

Some projects may use a different version of Node than your own. This can become a hassle to deal with, so what we would suggest doing is installing [Node Version Manager(nvm)](https://github.com/nvm-sh/nvm).

Another great alternative is [n](https://www.npmjs.com/package/n).

nvm allows us to pick which version of Node to use, install new versions, and more.

* [nvm (Node Version Manager)](https://github.com/nvm-sh/nvm) is a tool that allows you to download and install NodeJS and we use it whenever we want to get the latest Node version.
* [npm (Node Package Manager)](https://www.npmjs.com/get-npm) is a tool that allows you to install JavaScript packages. - *npm* comes with Node so if you have it installed you most likely have npm installed as well. - to check if you have it, run the following in the terminal: bash $ npm -v

**Mac OS**

Please follow these steps to successfully install Node on Mac OS:

1. install the *xcode*
2. install *nvm*
3. install *node*

**Linux OS**

Please follow these steps to successfully install Node on Linux OS:

1. Run the update and get the packages:

$ apt-get update

$ apt-get install build-essential libssl-dev

1. install *nvm*
2. install *node*

**Windows OS**

Please follow these steps to successfully install Node on Windows OS:

1. install *nvm* and *node*

To check if nvm and Node were installed, for any operating system, run the following:

* nvm:

$ nvm --version

# 0.32.1 (current, but in the time you check might be different)

* Node:

$ node --version # or node -v

# v12.6.0 (current, but in the time you check might be different)

To update Node,

* on Mac OS:

$ nvm install latest-stable-version

# (which you can find on the official NodeJS website: https://nodejs.org/en/)

* on Linux and Windows OS:

$ nvm install node

**Test if it works**

Let's create a new repository (my-first-node) and a new file in your that directory called test.js.

$ mkdir my-first-node

$ cd my-first-node

$ touch test.js

$ code .

In this file, add the following code:

console.log('Hello there! It seems your Node is installed!');

In your terminal, while inside the my-first-node repository, run the following:

$ node test.js

We can write any JavaScript code we like in Node, but we must keep in mind *there's no DOM*. That's why we will see the output in the terminal.﻿﻿  
So that's it! You have Node set up and you can dig in the backend web development.

Our first stop is to understand what is node package manager and what are these packages/libraries we can use thanks to NodeJS.

[Node Package Manager (npm)](https://www.npmjs.com/)

**What is a package?**

A package (also called *module* or *library*) is a reusable code which can be downloaded from a *global registry* into a developer’s local environment (read: on your machine). Each package may or may not depend on other packages.

In simple English, it's all about **code reusability** and not having to reinvent the wheel every time we need to repeat the same action.

Imagine the following scenario: You, as Mr./Mrs. Junior software developer, go to work at Megacorp Software after you graduate from Root Learn.

On your first day, a senior developer asks you to help build their shopping cart feature. They ask you to write the business logic for calculating shipping prices and tax for their new sports section.

You sit around for hours, researching logistics, shipping companies, and the tax rates for hundreds of cities around the world.

This is when you realize, "Hey, there are so many companies that use this feature as well, this functionality has to exist somewhere already." You actually can find most of the things you are looking for in the shopping cart exercise.

You, as the pro software developer, copy and paste *allllll* of the code to the sports shopping cart, making heavy use of your find and replace feature.

The next day, Mr(s). Senior developer asks you to build out *another* shopping cart, this time for food goods. Copy and paste the code again? No!

All of this functionality could be extracted into modules and shared across your codebase. The logic for calculating tax, logistics, shipping, all could be separate modules that everyone in the company shares!

*Not only is extracting code into modules a timesaver for you, but it also is a useful software development practice and saves the company time and money*

**What is a package manager, and why?**

According to [Wikipedia](https://en.wikipedia.org/wiki/Package_manager), a package manager or package management system is a collection of software tools that automates the process of installing, upgrading, configuring, and removing software packages for a computer's operating system in a consistent manner. It typically maintains a database of software dependencies and version information to prevent software mismatches and missing prerequisites.

Packages for your projects are hard to manage.

When working on your code amongst a team, it's *really* tricky to keep all the versions and dependencies aligned amongst everyone. Let's say Bob and you work on a project together. Bob is using version 1.2 of some package/module. You're using a version 1.8.

The problem here is that the code between those two versions may have changed drastically. Bob and you go to implement a new feature using this package, and now your version works on your computer, but not on Bob's.

Now imagine there are 100 other Bobs on your team. When that package updates, everyone has to update their version, or they're running into problems.  
  
This is a nightmare! However, package managers can come to rescue! One of them is the node package manager, known as npm.

**What is npm ?**

[*https://www.npmjs.com/*](https://www.npmjs.com/)*: npm official webpage might look familiar to you. 😉*

**npm** is a package manager for Node. It comes with Node.

**npm** allows you and other developers to share your JavaScript code easily, using a command-line tools.

npm has one of the most robust ecosystems for external packages. At the time of writing, there are over 836,000 libraries (npm modules) currently available.

npm encourages good software development principles and code sharing.

As mentioned before, npm has thousands of modules available for your use. These modules include functionality such as coloring text in your terminal, taking command-line input, getting Chuck Norris jokes, and even full-blown web frameworks!

We will be heavily using npm packages so prepare to get familiar with a whole bunch of them.

**How do I get access to the npm ?**

If you installed Node, chances are you already have npm installed!

Test this by running the following command in the terminal:

$ npm --version

# 6.11.2 (this is a current version)

If you receive this or some other version number, you're good to go!

**Install and use packages from npm**

To be able to use packages in your app, you have to go through two steps:

1. **install the package**:

$ npm install official-name-of-the-package

1. **require it** so your application knows where exactly you want to use it:

const name-we-give-to-package = require("official-name-of-the-package");

The first step will result in creating a ***manifest file***.

A *manifest file* is a file that lists out all the packages our program depends on to function. When installing packages using npm, this file is going to be called package.json

**package.json**

A short overview of *package.json*:

This file is used to give information to npm that allows it to identify the project as well as handle the project's dependencies. It can also contain other metadata such as a project description, the version of the project in a particular distribution, license information, even configuration data - all of which can be vital to both npm and to the end users of the package. The package.json file is normally located at the root directory of a Node.js project. -[nodejs.org-What is the file package.json?](https://nodejs.org/en/knowledge/getting-started/npm/what-is-the-file-package-json/)

💡 In simple English, package.json file will define all of our project packages and dependencies

Run the following commands in your terminal:

$ mkdir npm-getting-started

$ cd npm-getting-started

$ npm init --yes

**npm init --yes** initializes a [default *package.json*](https://docs.npmjs.com/creating-a-package-json-file#creating-a-default-packagejson-file) and it is a mandatory step whenever you are creating an app using Node. Read between the lines - if you plan to use at least one npm package, you will have to have *package.json*.

However, the flag --yes is optional, and if you skip it, this will prompt you with a bunch of different questions. Feel free to hit return until the last yes. Or simply use --yes and don't worry about it.

As expected, this created a file called package.json. This is a JSON formatted file that specifies all of the modules your project will use and other information about your project.

📝 Also, if you're not familiar with **JSON**, don't worry! You will be during this module.

Let's install our first module!

$ npm install chalk # or npm i chalk

npm install (npm i) goes to the npm servers and grabs a package called "chalk". Starting from Node version 5, the package gets automatically added as a dependency to our package.json.

Your *package.json* should look something along these lines. Notice the *dependencies* and *chalk* added there.

{

"name": "npm-getting-started",

"version": "1.0.0",

"description": "",

"main": "index.js",

"scripts": {

"test": "echo \"Error: no test specified\" && exit 1"

},

"keywords": [],

"author": "",

"license": "ISC",

"dependencies": {

"chalk": "^2.4.2"

}

}

👀 In older versions of Node, you would have to use the --save flag to save the package's name in your package.json (as in npm install --save chalk). If you see that in your searches, feel free to omit it.

Also, we can notice two new things in our main repo: *node\_modules* and *package-lock.json*. Hold on it for a second, and we will get back to it.

Let's use the *chalk* npm package now. When inside *npm-getting-started* repository, run the following:

$ touch index.js

$ code .

And then inside of index.js:

const chalk = require('chalk'); // this is the second step

// ("require package so your app knows where you want to use it")

console.log(chalk.blue('Hello, npm!'));

Firstly, we require the module and assign it to a variable.

Then, we can call upon this variable and its attached blue method. How do you know which methods are there to be used? Well, the documentation has all the answers. 😉

You can see all the functionality Chalk makes available by visiting [its documentation page](https://www.npmjs.com/package/chalk). A quality package has proper documentation for people to learn how to use it.

Now let's run index.js file and check the terminal to see the output:

$ node index.js

﻿﻿

🏆 How does package.json help us? As we said, whenever you collaborate with others, you have to be aligned on which version of every single package to use. **You should commit to GitHub, or some other code sharing platform, your package.json, so any user would be able to recreate your node\_modules by simply running npm install**.

Again we mentioned **node\_modules** so let's demystify it.

**node\_modules**

* **node\_modules** folder gets generated when we run npm initand when the package.json gets generated as well
* node\_modules will contain all the packages/libraries you install/download from the npm.
* node\_modules shouldn't be ever pushed to GitHub because:
  + it's a huge folder and no need to be pushed, as well as,
  + anyone who clones your repository will be able to regenerate it with npm install based on the package.json.

In simple English, node\_modules folder has locally, on your machine, all the libraries/packages you defined in *package.json*.

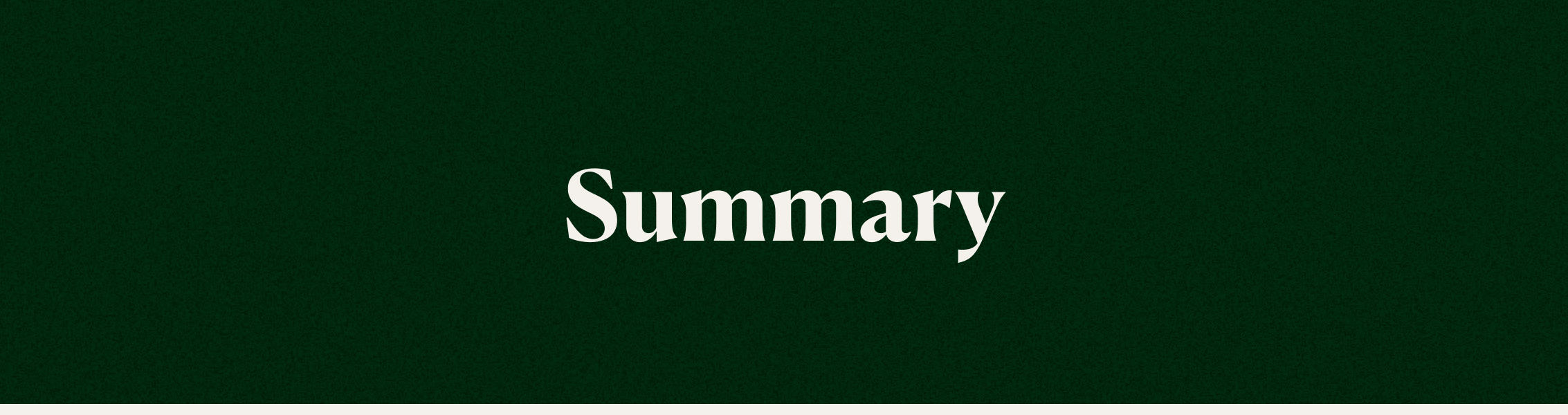
💣 **Never commit node\_modules to git**. It will make your repository a lot larger. Your node\_modules may also not work on another computer. Everyone has to install npm packages for themselves with npm install. Create a .gitignore file (if it does not already exist) and add node\_modules to it.

Reminder: The shorthand for npm install is just **npm i**. ⭐

**node\_modules** folder - this is where the actual code for the chalk module is stored. package-lock.json - the primary purpose of this file is to make sure that installing modules from the same package.json doesn't result in two different installs. How's this possible? Let's see.  
  
📝 **Time to practice**

Install a package from [this list of npm packages](https://github.com/sindresorhus/awesome-nodejs#weird), specifically the "weird" section.

Use the installed package in your index.js. See its documentation to learn what it can do and how to use it.



**Summary**

In this lesson we learned a bit about Node.js, it's history, and how to install it on our computer. Also, we learned a bit about what *runtime* is.

You don't need to know the history of Node to get started, but it is essential to understand the relationship between Node and JavaScript to avoid looking silly in interviews in the future.

❗ **It's important to remember that the Node is not a framework nor a language**.

**Extra Resources**

* [NodeJS - the official docs](https://nodejs.org/en/docs/)
* [The Javascript Runtime Environment - Medium post](https://medium.com/@olinations/the-javascript-runtime-environment-d58fa2e60dd0)
* [npm packages - semantic versioning](https://docs.npmjs.com/about-semantic-versioning)
* [Getting the best out of NPM](https://jsblog.insiderattack.net/getting-the-best-out-of-npm-b73d4a8fbf29#.hehmw1xun) - More advanced NPM concepts
* [What are package.json and package-lock.json?](https://dev.to/saurabhdaware/but-what-the-hell-is-package-lock-json-b04)