



TASK

NodeJS

Visit our website

Introduction

WELCOME TO THE NODEJS TASK!

NodeJS (or just Node) is a JavaScript runtime environment that works outside a web browser. More specifically, it is used in a server (rather than a client). Before Node, PHP was a common go-to for web applications. However, there were many downsides to PHP that Node was able to address. For example, real-time server interaction is impossible with PHP. Node also introduced many optimisations, which meant faster server executions. In this task, we go through the general installation of NodeJS and end off with a few of our own .js programs.



Get in touch
Connect for support

Remember that with our courses, you're not alone! You can contact an expert code reviewer to get support on any aspect of your course.

The best way to get help is to login to Discord at <https://discord.com/invite/hyperdev> where our specialist team is ready to support you.

Our team is happy to offer you support that is tailored to your individual career or education needs. Do not hesitate to ask a question or for additional support!





A note from the Hyperion Team

The Best Qualification for Programmers.

[This article](#) on the Hyperion Hub answers a question we get asked time and time again: “What degree should I study if I want to become a software developer?” We previously published an article on the high-level differences between Information Technology (IT) and Computer Science (CS). In this article, we compare and contrast the employability, salaries and career paths for IT versus CS qualifications and answer which qualification is the best for aspiring programmers.

IF YOU HAVE NOT ALREADY INSTALLED NODE:

FOR LINUX/WSL USERS

The easiest way to install software in Linux (specifically the Debian variations like Ubuntu) is using the **apt-get** package manager. To install NodeJS on Linux/WSL, you simply type in:

```
> sudo apt install npm nodejs
```

Let's break down what this command means:

1. **sudo**: this is placed at the beginning of any command that needs super-user privileges, such as this one. This means “super-user do”. If you aren't logged in as a super-user, this command will prompt you for your password
2. **apt**: run the apt-get application (apt is just short for apt-get).
3. **install**: use the install option within apt.
4. **npm**: this is the node package manager. This will be useful later on.
5. **nodejs**: the package to install. The list of installable packages can be seen with apt-cache search.

You can also follow these steps to install Node:

<https://heynode.com/tutorial/install-nodejs-locally-nvm/>

FOR WINDOWS USERS

Go to the NodeJS download page [here](#).

You will be presented with a set of download options that looks something like this:


Downloads

Latest LTS Version: **16.16.0** (includes npm 8.11.0)

Download the Node.js source code or a pre-built installer for your platform, and start developing today.

LTS

Recommended For Most Users




Windows Installer

node-v16.16.0-x64.msi


Current

Latest Features



macOS Installer

node-v16.16.0.pkg



Source Code

node-v16.16.0.tar.gz

Windows Installer (.msi)	32-bit	64-bit
Windows Binary (.zip)	32-bit	64-bit
macOS Installer (.pkg)	64-bit / ARM64	
macOS Binary (.tar.gz)	64-bit	ARM64
Linux Binaries (x64)	64-bit	
Linux Binaries (ARM)	ARMv7	ARMv8
Source Code	node-v16.16.0.tar.gz	

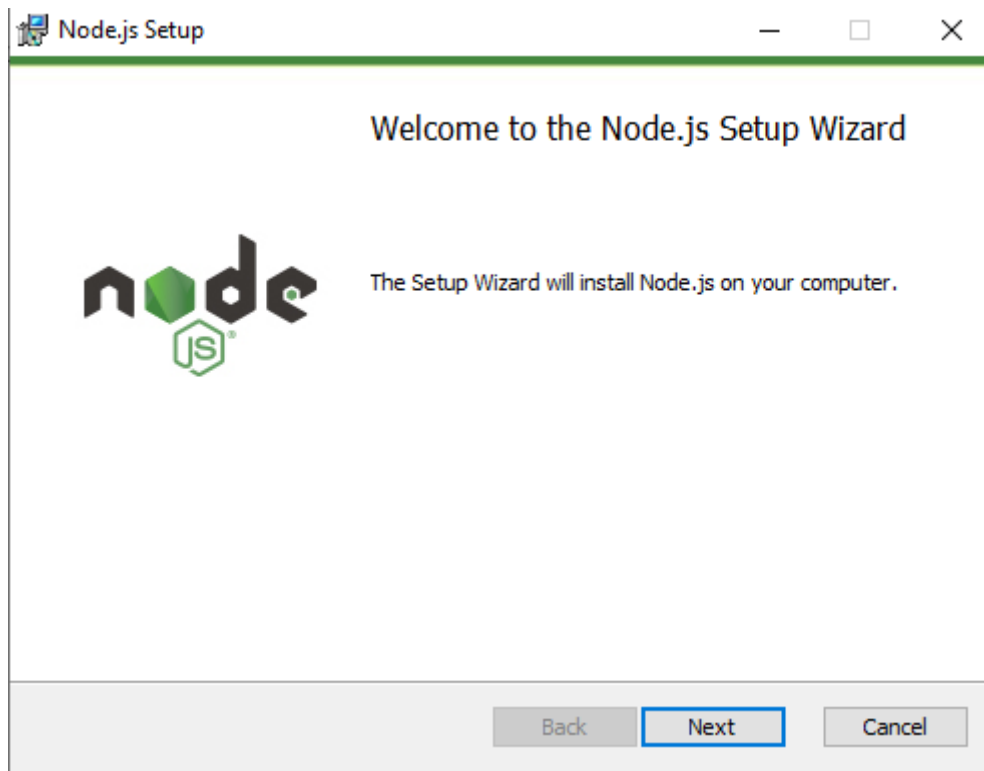
Because you are running Windows, you should choose the Windows Installer (.msi). Most of you will need to choose the 64-bit version. If you are running a 32-bit system, then you will need to use the 32-bit installer.

If you're unsure which system you're using currently, just click the **Start** button and then select **Settings**. Then click **System** and choose **About**. This should reveal the bit-version for your particular System. If you're using a Mac, then you'll find your processor type in **About This Mac**.

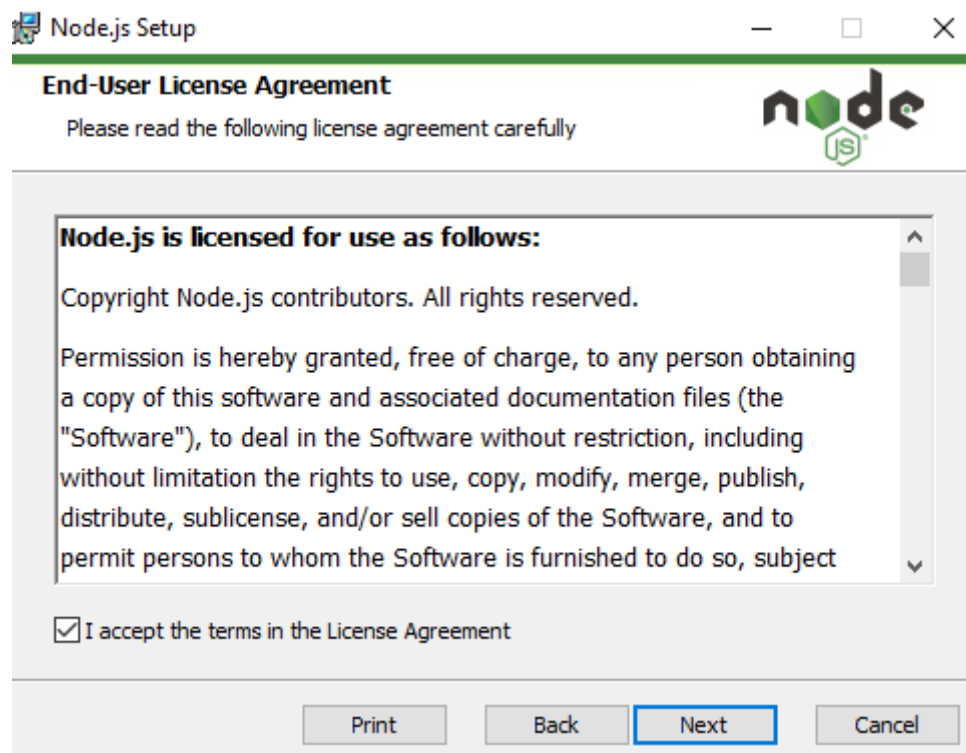
After downloading it, double-click the file to open it. You will be greeted with something that looks like this:

 HyperionDev

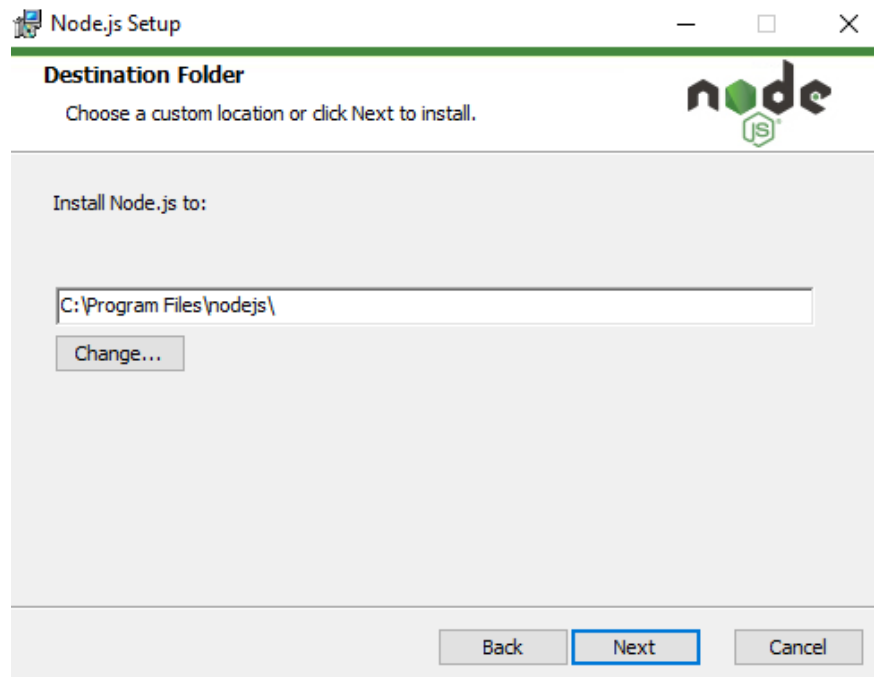
Copyright © 2021 HyperionDev. All rights reserved.



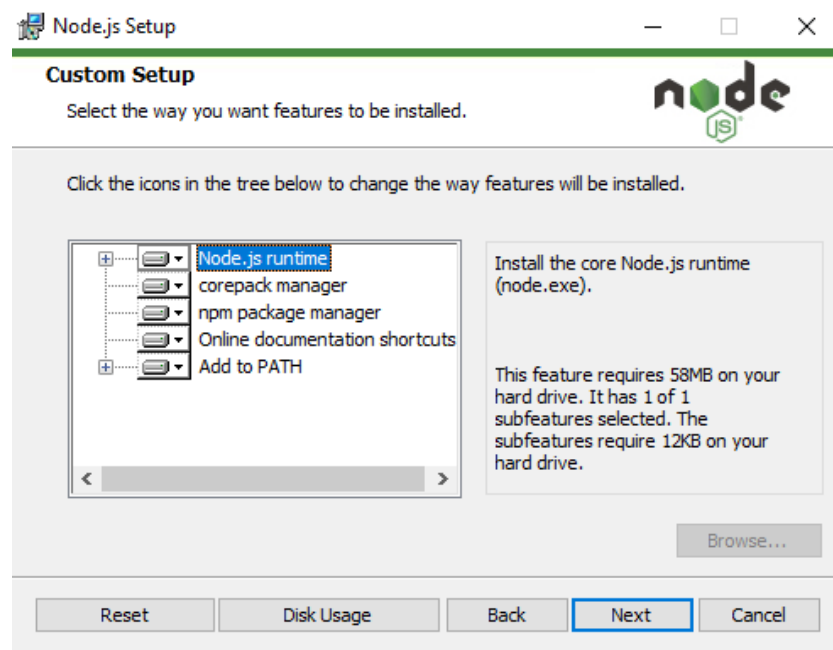
It will ask you to accept the EULA. Click **accept**, then click **Next**.



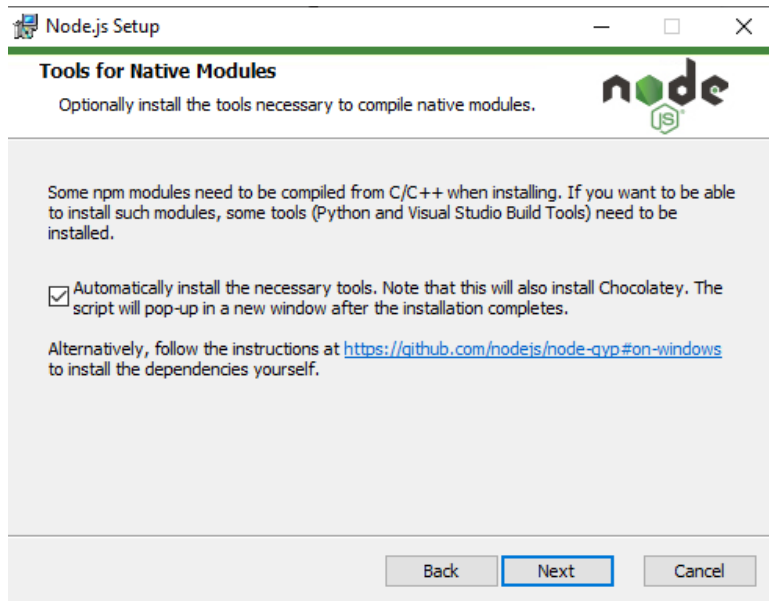
The next step is choosing **where** you would like to install Node. Select the location (or use the default) and click **Next**.



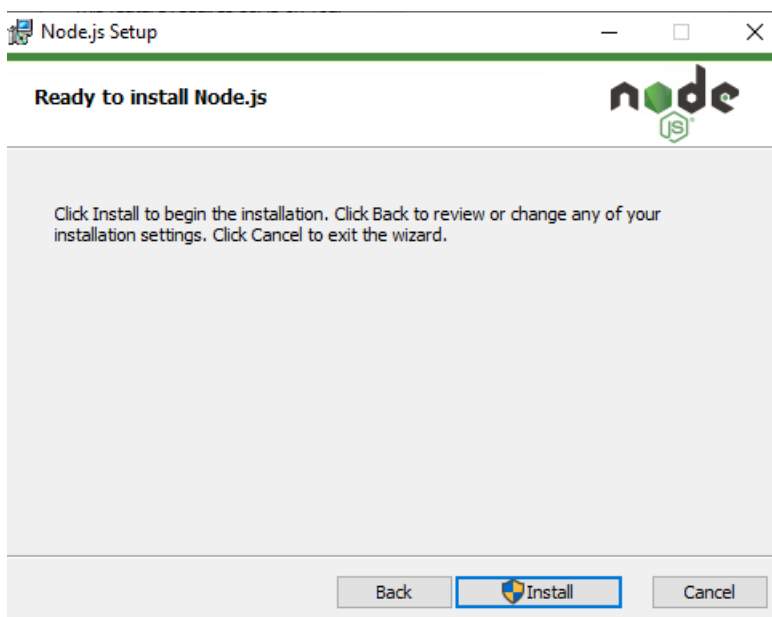
Once this is done, you'll be presented with a set of complex-looking options. Good news: these are just the defaults: click **Next**:



The final screen, before installation, will inform you that certain dependencies will need to be installed for NodeJS to run properly. It presents you with an option to auto-install them, **check the box** and click **Next**:



Now that your set-up is done, you can install Node! This will require some **admin privileges**:



FOR MAC USERS

Use the macOS in-built package manager: **brew**. To use this option, open a terminal. To do this, press Command+Space and type in "Terminal".

Follow the instructions : <https://formulae.brew.sh/formula/node>

Once your terminal is open, type in:

```
> brew install node
```

And Brew will install NodeJS for you. Simple!

HELLO NODE!

Now that you have your NodeJS installation set up, let's say: hello! Create a file called **helloworld.js** and put the following line of code in it:

```
console.log("Hello world!");
```

Then, to run your code, simply open up a command line or terminal, and type in:

```
> node helloworld.js
```

MANAGING NODE PACKAGES

Now that we know how to run a JavaScript file using Node, let's take a look at some practical applications. When you downloaded and installed Node to your system, you also included the **Node Package Manager** (NPM) along with it. Whenever you create an application in Node, NPM is included as a **module**. A module is an encapsulated set of code with a specific function. In order to achieve this, you will need to make use of packages, hence the package manager. Let's take a look at how we can achieve this.

Let's create our own module called *my_first_module*. To do this, open up a terminal or command line, and type:

```
> mkdir my_first_module  
> cd my_first_module
```

This creates a normal blank directory called *my_first_module* and the second line just changes the current working directory into that module. Now that we are in this directory, we need to initialise a new Node module using NPM:

```
> npm init
```

Then simply hit **enter** at each prompt for all of the defaults (unless there are specific things you want to change, but this won't be too important right now). You will now see a new file in the directory called *package.json*. This is a file that gives Node information about the package.

Congratulations! You have now created your first module! This won't be incredibly useful if you can't include any packages that are available for Node. Let's start with a common one: **lodash**. This package comes bundled with a lot of common utility functions that makes using JavaScript a lot easier. To add this package to your module, type in:


```
> npm install lodash
```

You will notice a new folder called *node_modules* after a long period of downloading. This is where all of the *lodash* code is downloaded and stored.

You will also notice a few changes now. In your *package.json* file, you will now see:

```
"dependencies": {  
  "lodash": "^4.17.21"  
}
```

Additionally, you will see a new file: *package-lock.json*. This file exists for better portability into different systems. Typically, you would not include all of the code from *node_modules* in your Git repository (due to the large amount of code necessary). To ensure that the correct version of code is downloaded each time round, the *package-lock.json* file specifies exact versions, download locations, and hashes. If you open up the *package-lock.json* file, you will see:

```
"dependencies": {  
  "lodash": {  
    "version": "4.17.21",  
    "resolved": "https://registry.npmjs.org/lodash/-/lodash-4.17.21.tgz",  
    "integrity":  
    "sha512-v2kDEe57lecTu1aDIuNTPy3Ry4gLGJ6Z103vE1krqXZNrsQ+LFTGHVxVjcXPs17LhbZVGedAJv8XZ1tvj5FvSg=="  
  }  
}
```

In order to use *lodash*, start by creating a JavaScript file. Then, within the file, include the following code:

```
const lodash = require('lodash');
```

This loads the *lodash* module, and allows you to use all of the code within it. For example:

```
myList = ["This", "is", "a", "list", "in", "Javascript"];  
console.log(lodash.first(myList));
```

USING THE INSTALLED PACKAGES: COMMONJS VS ES6

In NodeJS, there are two main module systems, which define how you import modules. The most commonly-used one in Node is quite aptly named **CommonJS**. The other one is called **ES6**. Regular web browsers have support for imports done

with ES6, whereas most common NodeJS modules were written with CommonJS export structures.

The main difference between them is how the import takes place: statically or dynamically. **CommonJS imports happen dynamically**, which means that they happen at **run-time**. CommonJS imports happen through the `require()` method. This has many different implications. For example, the name of the module can be determined at runtime and passed through a variable. Or, for example, the import can take place in an if-statement or a loop.

ES6 imports happen statically, which means that they happen during compile-time. Given that they occur statically, it isn't possible to place them within an if-statement or a loop, nor can you determine the module with a variable.

SETTING UP YOUR OWN SCRIPTS IN NODEJS

Sometimes, navigating file names can be confusing, especially when needing to input different arguments. NPM provides a handy tool to navigate this: **scripting**.

Recall the `package.json` file that was set up when you initialised your module. There are a few impressive features included in the file. One of which is using scripts. In fact, it comes bundled with one default script:

```
"scripts": {  
  "test": "echo \"Error: no test specified\" && exit 1"  
},
```

This is, admittedly, not very helpful. However, if you have set up a testing framework, this can be handy. To better understand how this works, let's make an example ourselves.

You can start by creating two files: `foo.js` and `bar.js`.

In `foo.js`, place the following code:

```
console.log("I am working right now.")
```

And in `bar.js`, place the following code:

```
console.log("Goodnight!")
```

Now, let's make a modification to our `package.json` file:

```
"scripts": {  
  "test": "echo \"Error: no test specified\" && exit 1",  
  "work": "node foo.js",
```

```
"sleep": "node bar.js"
},
```

Now, we have defined specific scripts for specific tasks. Now type the following into the terminal/command-line:

```
> npm run work
```

And you will get something like this:

```
> my_first_module@1.0.0 work /mnt/e/Hyperion/my_first_module
> node foo.js
```

I am working right now.

Similarly, if you type:

```
> npm run sleep
```

You will get:

```
> my_first_module@1.0.0 sleep /mnt/e/Hyperion/my_first_module
> node bar.js
```

Goodnight!

This can be a handy way to set up predefined scripts with easy-to-access names.

Instructions

First, read **example.js**. Open it using IDLE.

- **example.js** should help you understand some simple JavaScript. Every task will have example code to help you get started. Make sure you read all of **example.js** and try your best to understand the code contained in it.
- You may run **example.js** to see the output. Feel free to write and run your own example code before doing the Task.

Compulsory Task 1

Follow these steps:

- Create a program called **hello.js**. Create a variable called *myName*, which is a string containing your name
- Get the code to print out "Hello X", where X is your name.

Compulsory Task 2

Follow these steps:

- Create a program called **menu.js**.
- Within this file, create three variables *item1*, *item2*, and *item3*.
- Each of these variables are strings. For each variable, assign it one of your three favourite foods.
- Get the program to print out:
Order Confirmed:
 1. *item1*
 2. *item2*
 3. *item3*
- Where *item1*, *item2*, and *item3* are the three favourite food values in each variable.

Compulsory Task 3

Follow these steps:

- Create a module called **my_first_module**.
- Initialise the module with NPM.
- Install lodash to this module.
- Create a script called **remove_duplicates.js**.
- Within this script, you will need to import lodash, and use the [uniq](#) function.
- Create the following array:
[1, 2, 10, 100, 10, 2, 5, 6, 10, 1000, 7, 2, 100, 1, 5, 7, 10]
- Using lodash, print out that same array, but with all duplicates removed.
- Finally, set up your module to run the script using:
`> npm run rdup`

Once you are ready to have your code reviewed, **delete** the **node_modules** folder (please note that this folder typically contains hundreds of files which, if you're working directly from Dropbox, has the potential to **slow down Dropbox sync and possibly your computer**), compress your project folder and add it to the relevant task folder in Dropbox.

Optional Bonus Task

Follow these steps:

- Create a file called **pros_cons.txt**.
- Answer the following questions:
 - What is the difference between using CommonJS and ES6 for loading modules?
 - List 3 pros of using CommonJS, and list 3 cons of using CommonJS.
 - List 3 pros of using ES6, and list 3 cons of using ES6.



Rate us

Share your thoughts

Hyperion strives to provide internationally-excellent course content that helps you achieve your learning outcomes.

Think that the content of this task, or this course as a whole, can be improved, or think we've done a good job?

[Click here](#) to share your thoughts anonymously.

