GLAB 318.1.1:

NodeJS Installation and Basics

Version 1.0, 07/07/23
[Click here to open in a separate window.](#)

Introduction

NodeJS (or simply Node) is a very popular JavaScript runtime that enables developers to create JavaScript applications outside of the context of a web browser. This lab will walk you through the steps of installing Node and the Node Package Manager (npm) on Windows.

Objectives

- Install Node.
- Install npm.
- Use basic Node commands to run JavaScript code outside of a browser.
- Use basic npm commands to:
  - Create a package file.
  - Install packages.
  - Run a program.

Equipment

- A Windows-based computer with **administrator privileges.**
  - If you do not have administrator privileges, speak with your instructor about potential alternative solutions.

Submission

Submit your completed lab using the **Start Assignment** button on the assignment page in Canvas.

Your submission should include:

- A link to the GitHub repository for your project.
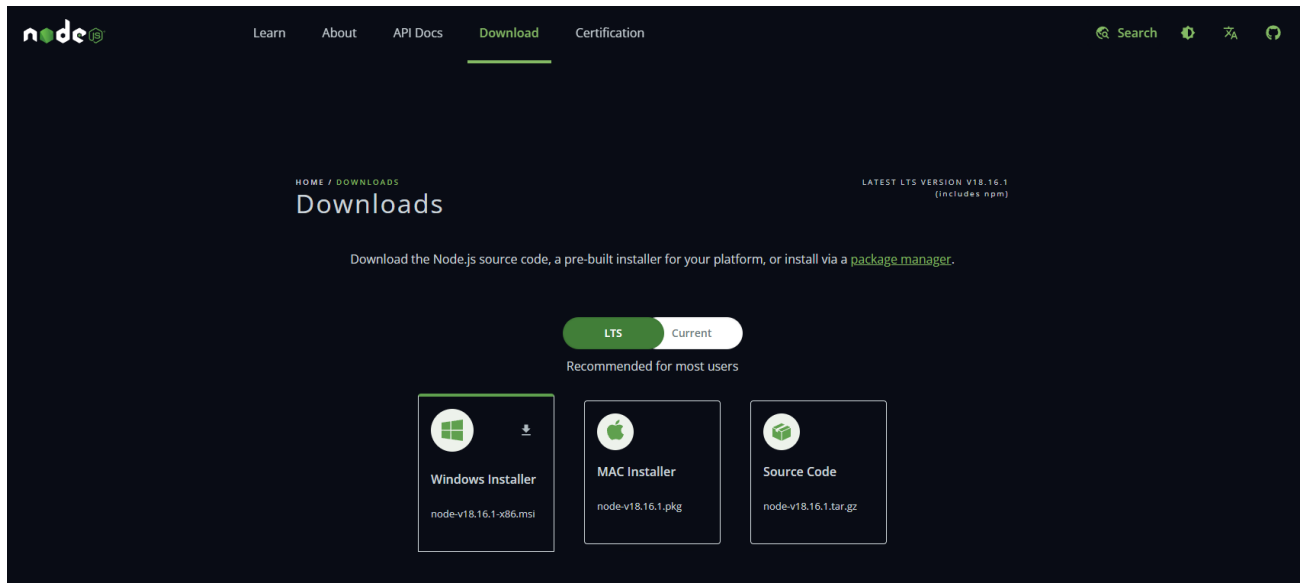
Instructions

We will begin by installing Node locally, and then work through common Node commands.

Part 1: Node Installation

Navigate to the [NodeJS downloads page](#).

On the downloads page, you should see installers for various operating systems. Download the Windows installer (or the installer that matches your operating system) for the **long-term support (LTS)** version of Node, and run the file once the download has completed.

Due to frequent version updates, the version of Node that you install may differ from what is pictured below. This is okay! Download the latest LTS version.


Note: if you are using an operating system other than Windows, the following screens and instructions may differ. Consult your instructor for clarification on the process for your specific operating system, where necessary.
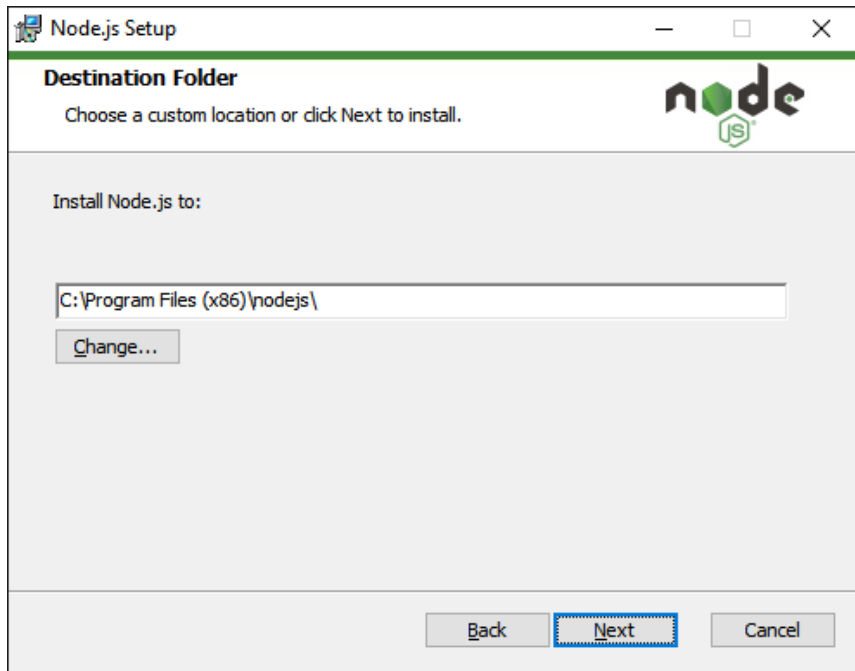
The NodeJS setup window should appear once the download has completed and the associated file has been run. Click **"Next."**
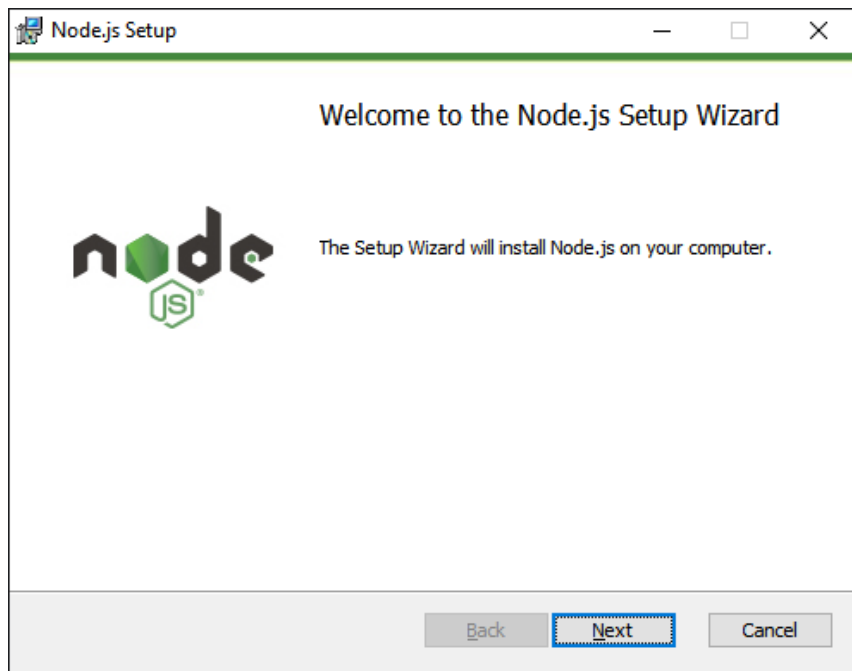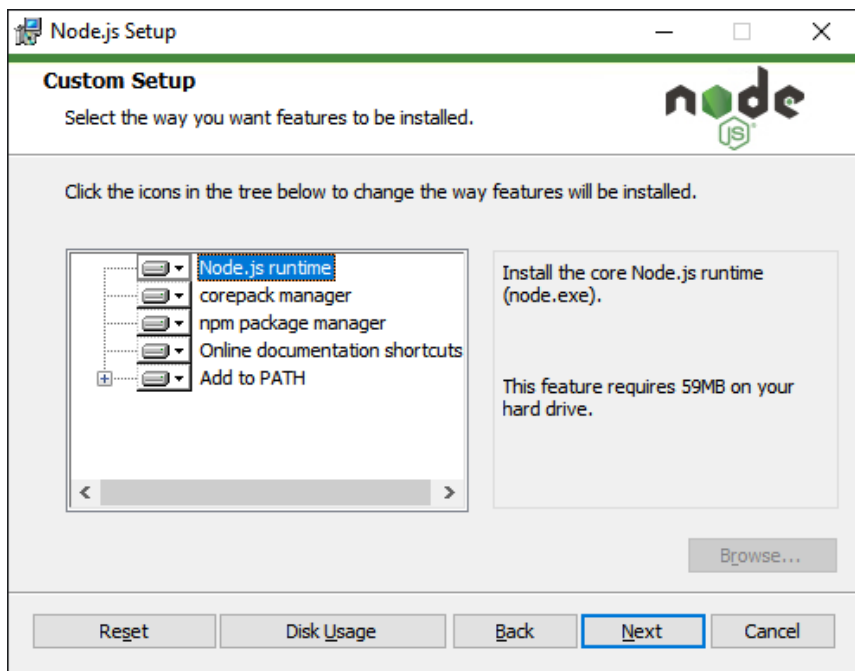
Read and accept the license agreement. Click **"Next."**
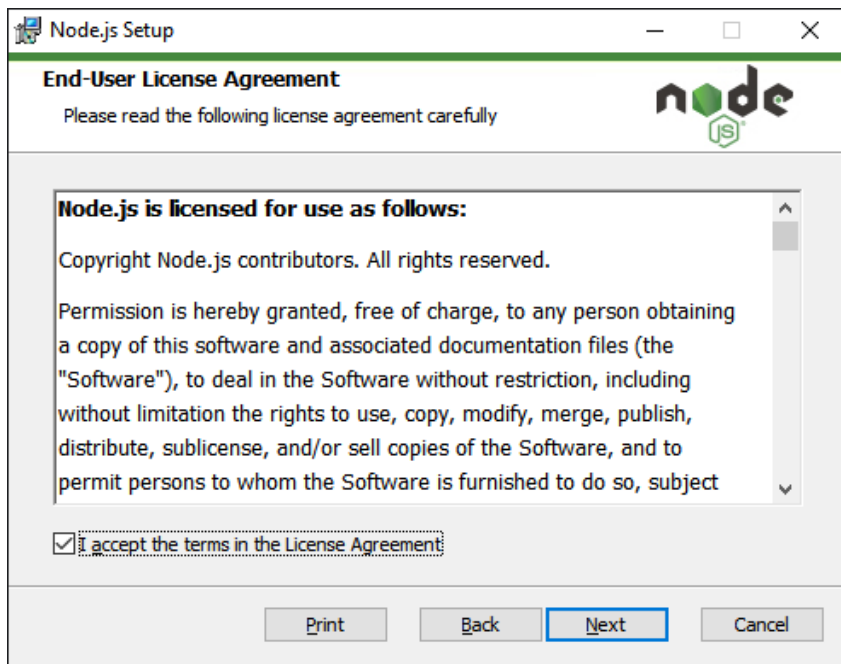
If you prefer, select a custom download location for Node. Otherwise, leave the default values and click **"Next."**

The following screen shows the different items that will be installed. Leave these values as their defaults and click **"Next."**

The **Tools for Native Modules** screen allows you to install some tools necessary for specific use cases. Unless your instructor has indicated that you should install these tools, you will likely not need them (and can always install them at a later date). Leave the default, and click **"Next."**
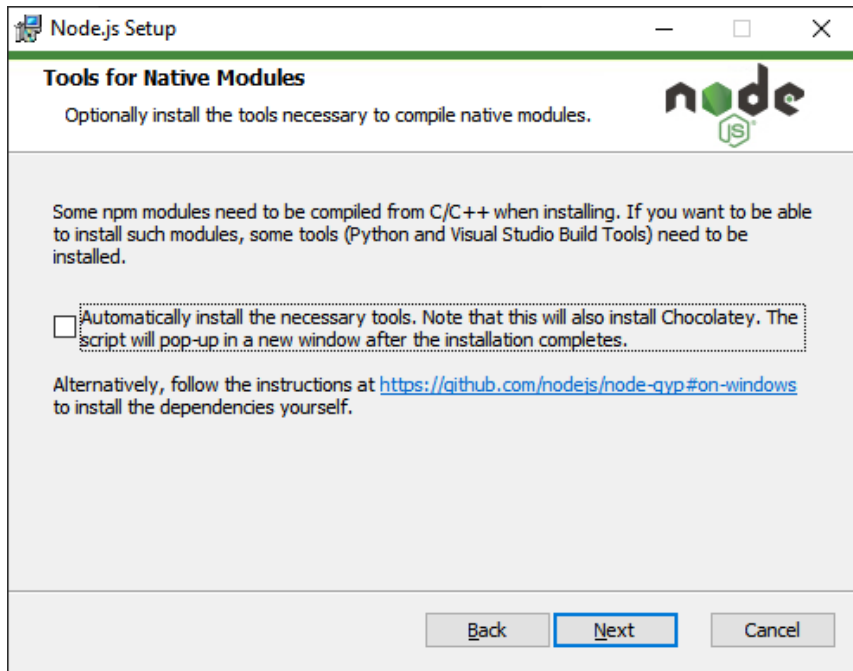
Finally, click **"Install."**
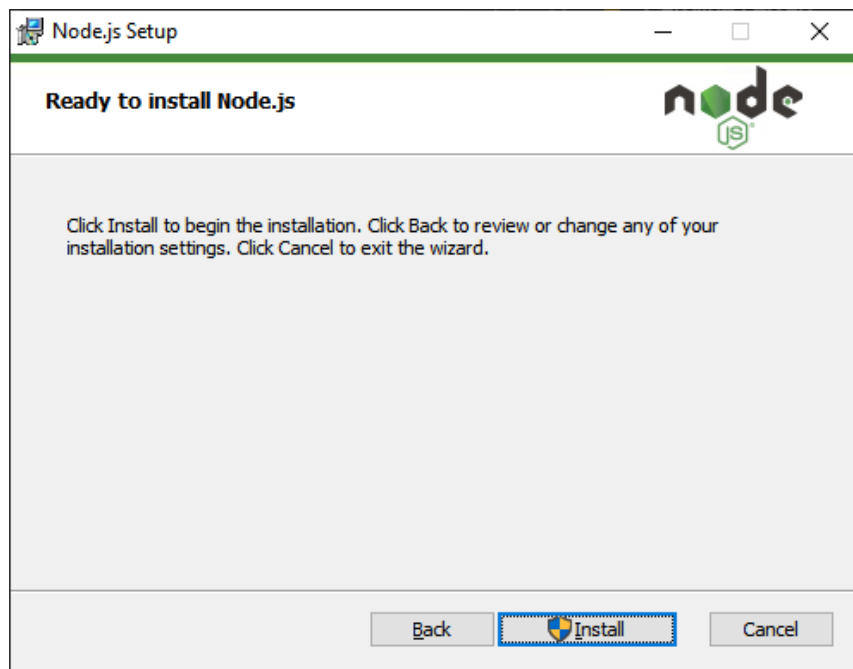
Once the installation has completed, move on to the next section.

Part 2: Node Command Line Interface

Open your command line tool of choice. We will be using Git Bash.

Use the version (-v) commands to verify installation of Node and npm:

- `node -v`
- `npm -v`

```
MINGW64:/c/Users/Matt/Documents/Per Scholas/Module 318        —  □  X

Matt@Matt-PC MINGW64 ~/Documents/Per Scholas
$ cd "Module 318"

Matt@Matt-PC MINGW64 ~/Documents/Per Scholas/Module 318
$ node -v
v20.3.1

Matt@Matt-PC MINGW64 ~/Documents/Per Scholas/Module 318
$ npm -v
9.6.7

Matt@Matt-PC MINGW64 ~/Documents/Per Scholas/Module 318
$
```

If your installation did not complete correctly, return to the steps in Part 1 and consult your instructor for aid. This is most likely due to issues with administrator privileges and the PATH variable. If your versions printed correctly, continue.

Next, we will run some very simple JavaScript code from the command line using the "eval" and "print" commands. The --eval or -e flag tells Node to evaluate the following string as a JavaScript expression. The --print or -p flag behaves similarly, but automatically prints the result if possible.

Use these commands to print the results of the expression "17 + 25," as follows:


```
MINGW64:/c/Users/Matt/Documents/Per Scholas/Module 318        —  □  X

Matt@Matt-PC MINGW64 ~/Documents/Per Scholas/Module 318
$ node -e 'console.log(17 + 25)'
42

Matt@Matt-PC MINGW64 ~/Documents/Per Scholas/Module 318
$ node -p '17 + 25'
42

Matt@Matt-PC MINGW64 ~/Documents/Per Scholas/Module 318
$ |
```

Congratulations, you have successfully run JavaScript code outside of the browser!

Of course, this is far from the extent of Node's capabilities. Create a file named "index.js" and save it in your current working directory (or navigate to a more convenient one using the change directory (cd) command.

Inside of the new JavaScript file, write some arbitrary code. We will use the classic "Hello World" example. Ensure that your file has some type of output using console.log!

Now, execute the file from the command line, as follows:


```
MINGW64:/c/Users/Matt/Documents/Per Scholas/Module 318        —  □  X

Matt@Matt-PC MINGW64 ~/Documents/Per Scholas/Module 318
$ node index.js
Hello World

Matt@Matt-PC MINGW64 ~/Documents/Per Scholas/Module 318
$ |
```

This is now beginning to scratch the surface of how Node works. You can use Node to execute a JavaScript file, which can be part of a larger program that includes many other files, packages, and
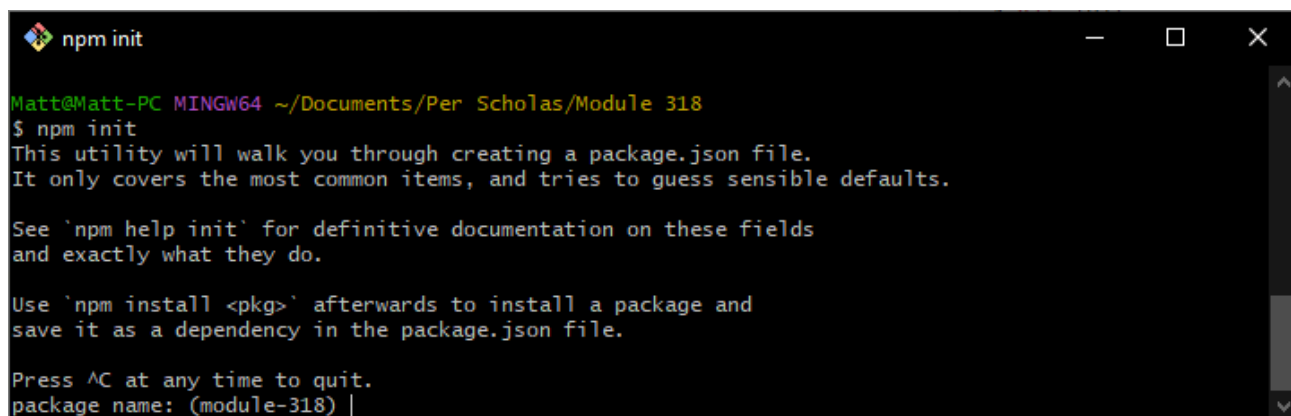
resources.

Part 3: Node Package Manager

The Node package manager, npm, is one of the most important tools for building Node applications.

Here's a fun fact from the official documentation that is worth knowing:

> Contrary to popular belief, npm is not in fact an acronym for "Node Package Manager"; it is a recursive bacronymic abbreviation for "npm is not an acronym" (if the project was named "ninaa," then it would be an acronym). The precursor to npm was actually a bash utility named "pm," which was the shortform name of "pkgmakeinst" - a bash function that installed various things on various platforms. If npm were to ever have been considered an acronym, it would be as "node pm" or, potentially "new pm."

To get started with some of the basic functions of npm, use the "init" command in the same directory that you previously created your test index.js file. This will give you the following screen:

As this describes, npm will attempt to walk you through creating a package.json file for your project. This file contains information used to identify the project and handle dependencies.

When starting a project from scratch, this is one of the first steps toward creating the appropriate configurations (you are likely already familiar with another one of the steps – git init – which creates the git repository within the project).

Here's how we have chosen to configure this sample project:

- **package name:** sample-project
- **version:** 1.0.0
- **description:** A sample project demonstrating the functionality of Node and npm.
- **entry point:** index.js

The remaining values can be left default or empty.

This is the final result:

```
MINGW64:/c/Users/Matt/Documents/Per Scholas/Module 318                    —    □    ✕

Matt@Matt-PC MINGW64 ~/Documents/Per Scholas/Module 318
$ npm init
This utility will walk you through creating a package.json file.
It only covers the most common items, and tries to guess sensible defaults.

See `npm help init` for definitive documentation on these fields
and exactly what they do.

Use `npm install <pkg>` afterwards to install a package and
save it as a dependency in the package.json file.

Press ^C at any time to quit.
package name: (module-318) sample-project
version: (1.0.0)
description: A sample project demonstrating the functionality of Node and npm.
entry point: (index.js)
test command:
git repository:
keywords:
author: Me
license: (ISC)
About to write to C:\Users\Matt\Documents\Per Scholas\Module 318\package.json:

{
  "name": "sample-project",
  "version": "1.0.0",
  "description": "A sample project demonstrating the functionality of Node and npm.",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "Me",
  "license": "ISC"
}


Is this OK? (yes)

Matt@Matt-PC MINGW64 ~/Documents/Per Scholas/Module 318
$ |
```

Note that the values given in parentheses can be accepted simply by hitting enter, and not all fields require information. Before creating the package file, npm will show you the exact structure of the JSON that will be written. You can verify this by navigating to the file in your working directory and inspecting its contents.

You can also have npm generate a default package file using the -y flag: npm init -y. You can always reconfigure your package file at a later date.

One of the useful functions of the package file is the ability to assign alias commands to specific tasks. We will create a "start" command that runs our project:

- Open the package.json file.
- Within the "scripts" object, add a new entry under "test" called "start."
- Set the value of "start" to "node index.js."
- Save the file.

Now, use the command "npm start" to start the program that you had previously created. Your program should run:

```
MINGW64:/c/Users/Matt/Documents/Per Scholas/Module 318          —    □    ✕

Matt@Matt-PC MINGW64 ~/Documents/Per Scholas/Module 318
$ npm start

> sample-project@1.0.0 start
> node index.js

Hello World

Matt@Matt-PC MINGW64 ~/Documents/Per Scholas/Module 318
$ |
```

While this may seem like a lot of work to end up writing "npm start" instead of "node index.js," these alias commands are required by a number of platforms for programs to run consistently. If a host platform uses the "npm start" command to run a program, it does not need to know what the name of your main file is. This means that we could change "index.js" to "application.js" if we wanted to, and it would not break anything.

These alias commands are also used for a variety of other purposes such as testing, debugging, development mode, deployment, etc.

As an example of this, let's explore the most important feature of npm: package installation.

The npm install (or simply npm i) command can be used to install any Node package on the npm network (there are more than *two million* at the time of this writing). Some packages, like the one we are about to install, are used in almost every application.

Use the following command:

- `npm i --save-dev nodemon`

nodemon is a tool that helps develop Node applications by automatically restarting the application when file changes are detected. This speeds up development efficiency considerably, which is why this package has so many downloads, as seen on the npm package page for nodemon.

The --save-dev flag tells npm that this is a development dependency, and does not need to be included in the production application.

```
MINGW64:/c/Users/Matt/Documents/Per Scholas/Module 318          —    □    ✕

Matt@Matt-PC MINGW64 ~/Documents/Per Scholas/Module 318
$ npm i --save-dev nodemon

added 33 packages, and audited 34 packages in 3s

3 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities

Matt@Matt-PC MINGW64 ~/Documents/Per Scholas/Module 318
$
```
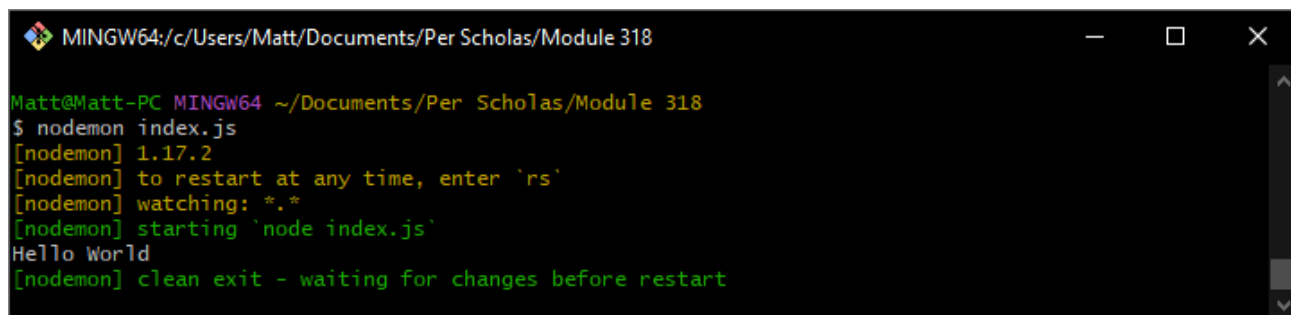
If you navigate to your project's directory, you will see a new folder called "node_modules" that contains all of the packages required to use nodemon. Within package.json, you will also notice a new "devDependencies" section that includes nodemon and the version number.
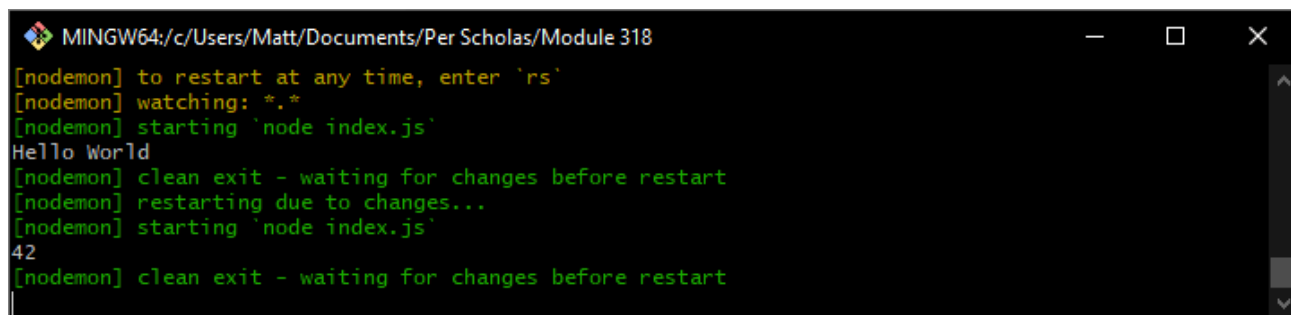
Using nodemon to run your program is simple:

- `nodemon index.js`

Note how the command line remains in an active state; it is not waiting for us to give it another command. Rather, nodemon is actively watching the program's files for changes.

Try modifying the contents of your index.js file, and save it. Watch how the command line responds.

Installing and putting a package to use with npm is that simple!

The convenience of npm for package management and installation cannot be understated; it is one of the most important tools for Node developers. In addition to nodemon, there are many other popular packages available on npm.

Below is a list of some of the most popular npm packages. You may already be familiar with some of these such as Axios, and others that will be introduced to you in the coming lessons. As you continue learning, explore the packages available to you!

Additionally, here is a reference for all available npm CLI commands.

The descriptions below, where available, have been provided directly from the npm package Readme.

- Express
  - Fast, unopinionated, minimalist web framework for Node.js.
- Async
  - Async is a utility module which provides straight-forward, powerful functions for working with asynchronous JavaScript. Although originally designed for use with Node.js and installable via npm i async, it can also be used directly in the browser.
- Lodash
  - Lodash makes JavaScript easier by taking the hassle out of working with arrays, numbers, objects, strings, etc.
- Axios

- ○ Promise based HTTP client for the browser and node.js
- [Karma](#)
  - ○ A simple tool that allows you to execute JavaScript code in multiple real browsers. The main purpose of Karma is to make your test-driven development easy, fast, and fun.
- [Mocha](#)
  - ○ Simple, flexible, fun JavaScript test framework for Node.js & The Browser
- [Moment](#)
  - ○ A JavaScript date library for parsing, validating, manipulating, and formatting dates.
- [Babel](#)
  - ○ Babel is a JavaScript compiler.
- [Socket.io](#)
  - ○ Socket.IO enables real-time bidirectional event-based communication.
- [Mongoose](#)
  - ○ Mongoose is a MongoDB object modeling tool designed to work in an asynchronous environment.
- [React](#)
  - ○ React is a JavaScript library for creating user interfaces.
  - ○ The react package contains only the functionality necessary to define React components. It is typically used together with a React renderer like react-dom for the web, or react-native for the native environments.
- [Redux](#)
  - ○ Redux is a predictable state container for JavaScript apps.
- [Jest](#)
  - ○ Delightful JavaScript Testing.
- [GraphQL](#)
  - ○ The JavaScript reference implementation for GraphQL, a query language for APIs created by Facebook.
- [Nodemailer](#)
  - ○ Send emails from Node.js – easy as cake!
- [dotenv](#)
  - ○ Dotenv is a zero-dependency module that loads environment variables from a .env file into process.env. Storing configuration in the environment separate from code is based on The Twelve-Factor App methodology.
- [Passport](#)
  - ○ Passport is Express-compatible authentication middleware for Node.js.
- …and literally millions more!

Part 4: Creating a Server

Node's purpose is to allow developers to create back-end systems to pair with their application's front-end. In order to do this, we need to develop a server that listens for communication from the client and returns a response.

Empty the contents of your index.js file, and follow along with the code below.

First, we want to use the require keyword to include the http module, which is built into Node (and does not need to be installed separately by npm). This module allows node to transfer data over the

Hyper Text Transfer Protocol (HTTP) used by the internet.

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

Next, we need to define the location and port of the server. For now, we will use a local address; there are better ways to handle this, which we will explore in future lessons.

```
const hostname = '127.0.0.1';
const port = 3000;
```

The createServer method of the http object allows us to define how the server will behave. The configuration below is extremely simple for example purposes, and will only ever respond with a status code of 200 (meaning "success") and the text content "Hello World!"

The variables req and res correspond to the request and response objects of the server communications. Each of these has a variety of properties and methods that we will explore in future lessons. These abbreviations are used almost everywhere, so you should familiarize yourself with them.

```
const server = http.createServer((req, res) => {
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain');
  res.end('Hello World!\n');
});
```

Finally, we need to tell the server to actually *listen* for communications. We do this by calling the listen method of the server object we just created, and passing it the port and hostname we previously defined. It also accepts a callback function that we can use to indicate that the server is running.

```
server.listen(port, hostname, () => {
  console.log(`Server running at http://${hostname}:${port}/`);
});
```

Save the file, and use nodemon to start the application.

Open a web browser, and navigate to 127.0.0.1:3000 (or localhost:3000).

You should see the response being sent from the server: Hello World!

Congratulations, you just created your first Node server! Time permitting, feel free to explore the other options available to you within your Node server. For example, what happens if we change our createServer call to the following? Try it!

```
const server = http.createServer((req, res) => {
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/html');
  res.write('<h1 style="color: red">Hello World!</h1>');
  res.write('<p>I wonder what else we can send...</p>');
  res.end();
});
```

Part 5: Open Exploration

Using what you have learned about HTTP request routing, create at least two different routes for your application. Have the "default" route continue to render the content above.

Within those custom routes, explore your options! Be creative, and see what you can build by writing HTML to the response, reading and parsing different portions of the request, and implementing specific logic to create a unique experience.

This lab is not graded, so do not worry if your routes end up being silly, impractical, inefficient, or ineffective. This time is for you to explore the new possibilities of using JavaScript outside of the context of a web browser.

Use whatever time is available to continue your exploration. Brainstorm with your peers, reference documentation, and research the possibilities available to you.

When you begin running out of time, make sure that you **comment out any code that prevents the program from running**.

Part 6: Completion

Upload your project to a GitHub repository, and submit it according to the submission instructions at the beginning of this document.