Agenda

- What is Node.Js? What is a multipage application?
- Installation of Node.Js and NPM
- npm tool, Yarn and npx
- Create a node project.
- npm init
- Project execution
- Package.json
- Modules
- What are modules? Types of modules.
- Main modules: HTTP, FS, OS, Path and URL.
- Process.agrv property

What is Node.Js ?

Node.js is an open-source and cross-platform JavaScript runtime environment. It is a popular tool for almost any kind of project!

Node.js runs the V8 JavaScript engine, the core of Google Chrome, outside of the browser. This allows Node.js to be very performant.

A Node.js app runs in a single process, without creating a new thread for every request. Node.js provides a set of asynchronous I/O primitives in i standard library that prevent JavaScript code from blocking and generally, libraries in Node.js are written using non-blocking paradigms, making blockir behavior the exception rather than the norm.

When Node.js performs an I/O operation, like reading from the network, accessing a database or the filesystem, instead of blocking the thread ar wasting CPU cycles waiting, Node.js will resume the operations when the response comes back.

This allows Node.js to handle thousands of concurrent connections with a single server without introducing the burden of managing thread concurrenc which could be a significant source of bugs.

Node.js has a unique advantage because millions of frontend developers that write JavaScript for the browser are now able to write the server-side coc in addition to the client-side code without the need to learn a completely different language.

In Node.js the new ECMAScript standards can be used without problems, as you don't have to wait for all your users to update their browsers - you a in charge of deciding which ECMAScript version to use by changing the Node.js version, and you can also enable specific experimental features b running Node.js with flags.

What is a Multi-Page application ?

Multiple-page applications work in a “traditional” way. Every change eg. display the data or submit data back to server requests rendering a new pag from the server in the browser. These applications are large, bigger than SPAs because they need to be. Due to the amount of content, thes applications have many levels of UI. Luckily, it’s not a problem anymore. Thanks to AJAX, we don’t have to worry that big and complex applications hav

to transfer a lot of data between server and browser. That solution improves and it allows to refresh only particular parts of the application. On the oth hand, it adds more complexity and it is more difficult to develop than a single-page application.

Pros of the Multiple-Page Application:

- It's the perfect approach for users who need a visual map of where to go in the application. Solid, few level menu navigation is an essential part of traditional Multi-Page Application.
- Very good and easy for proper SEO management. It gives better chances to rank for different keywords since an application can be optimized for one keyword per page.

Cons of the multiple-page application:

- There is no option to use the same backend with mobile applications.
- Frontend and backend development are tightly coupled.
- The development becomes quite complex. The developer needs to use frameworks for either client and server side. This results in the longer time of application development.

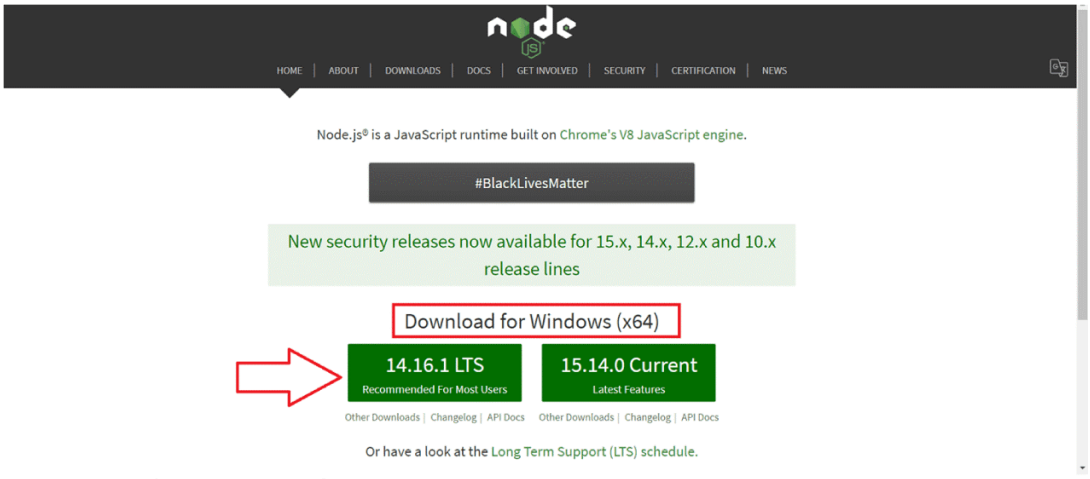
Installation of Node.Js and NPM:

Here, we are going to explain the installation process step-by-step. So, let's start with the first step now.

Step 1: Download the Installer

Download the **Windows Installer** from **NodeJs official website**. Make sure you have downloaded the latest version of NodeJs. It includes the NP package manager.

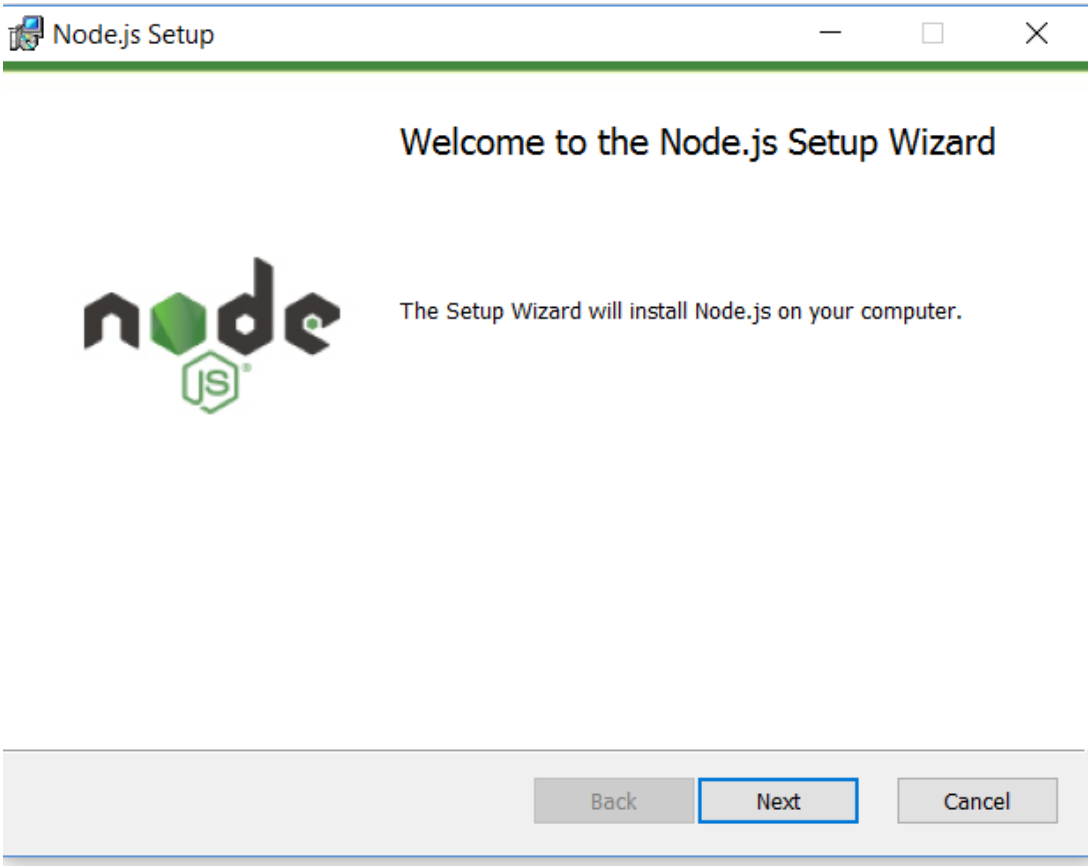
Here, we are choosing the 64-bit version of the Node.js installer.



The LTS (Long-term Support) version is highly recommended for you. After the download of the installer package, install it with a double-click on it.

Now .msi file will be downloaded to your browser. Choose the desired location for that.

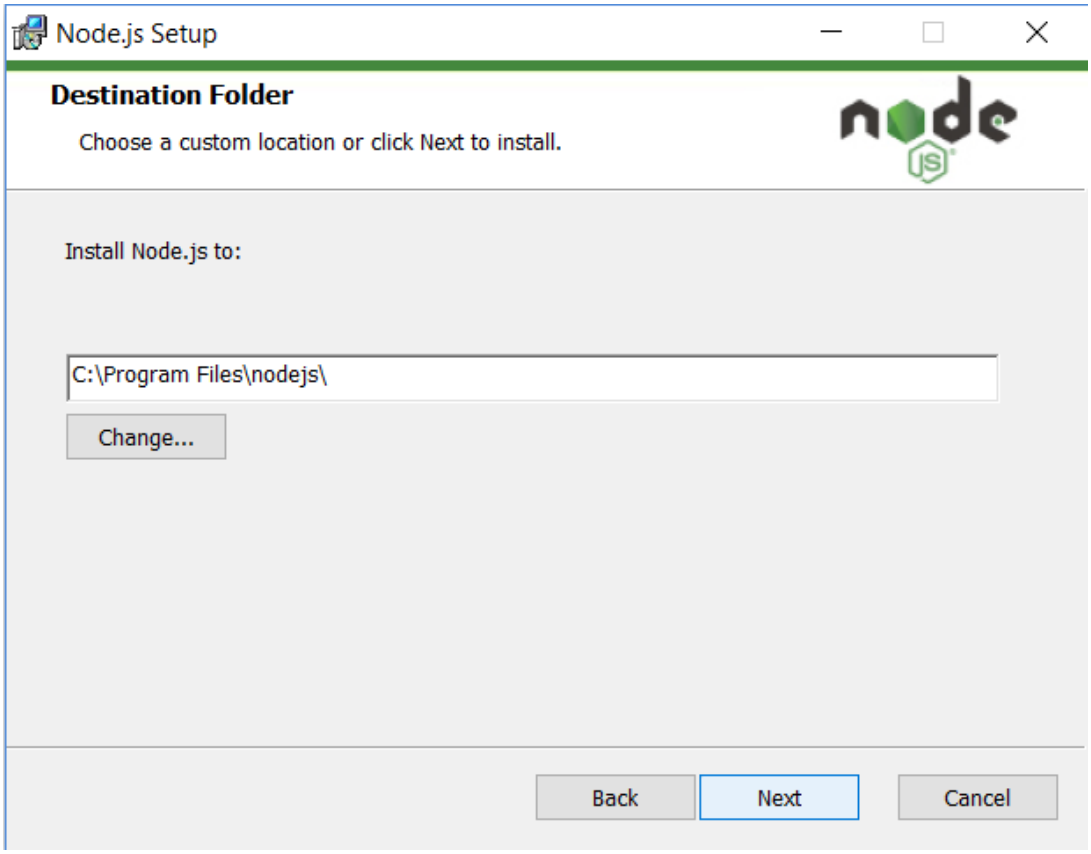
Step 2: Install Node.js and NPM



After choosing the path, double-click to install .msi binary files to initiate the installation process. Then give access to run the application.

You will get a welcome message on your screen and click the “Next” button. The installation process will start.

- Choose the desired path where you want to install Node.js.

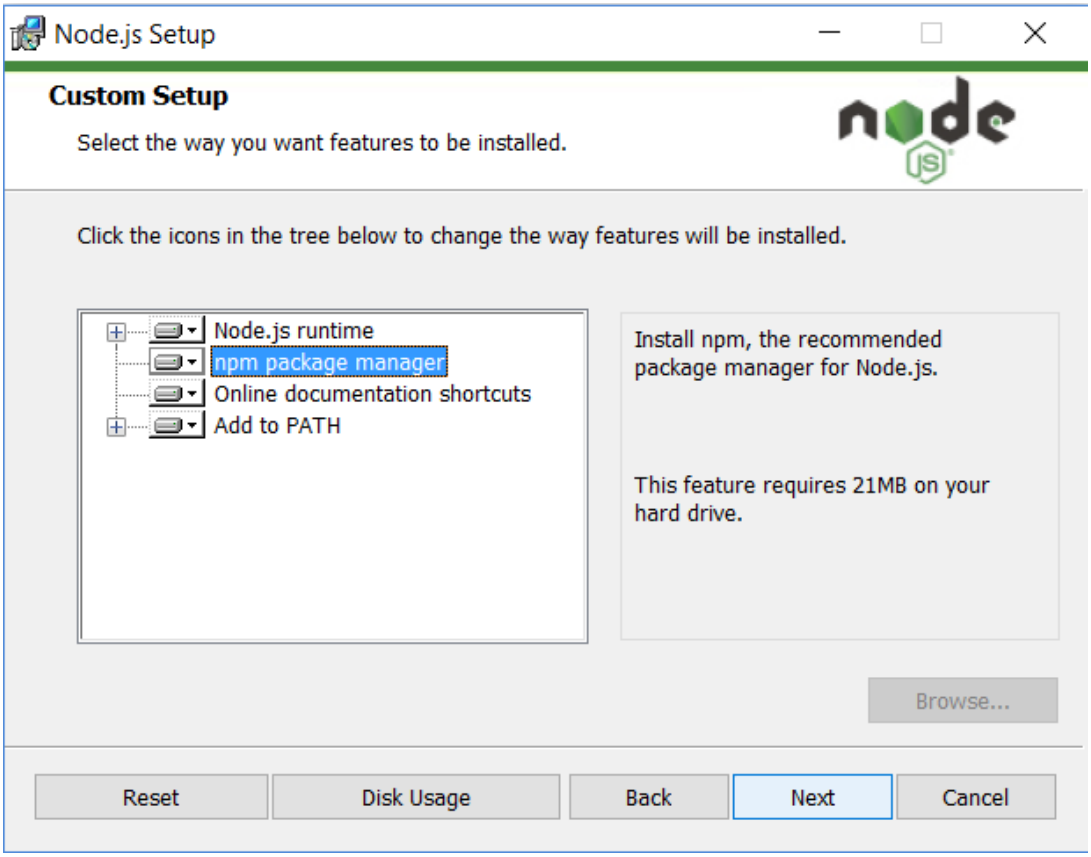


- By clicking on the Next button, you will get a custom page setup on the screen. Make sure you choose **npm package manager** , not the default of **Node.js runtime** . This way, we can install Node and NPM simultaneously.

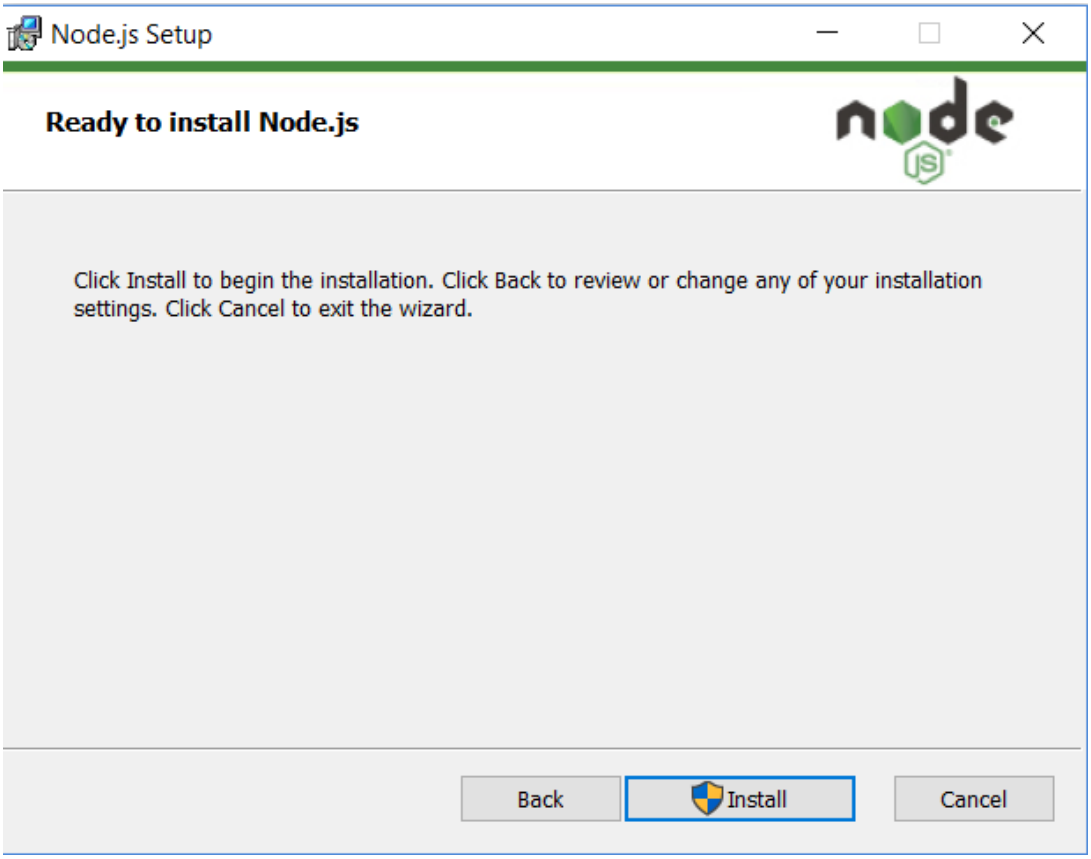
You should have 143MB of space to install Node.js and npm features.

The following features will be installed by default:

- Node.js runtime
- Npm package manager
- Online documentation shortcuts
- Add to Path

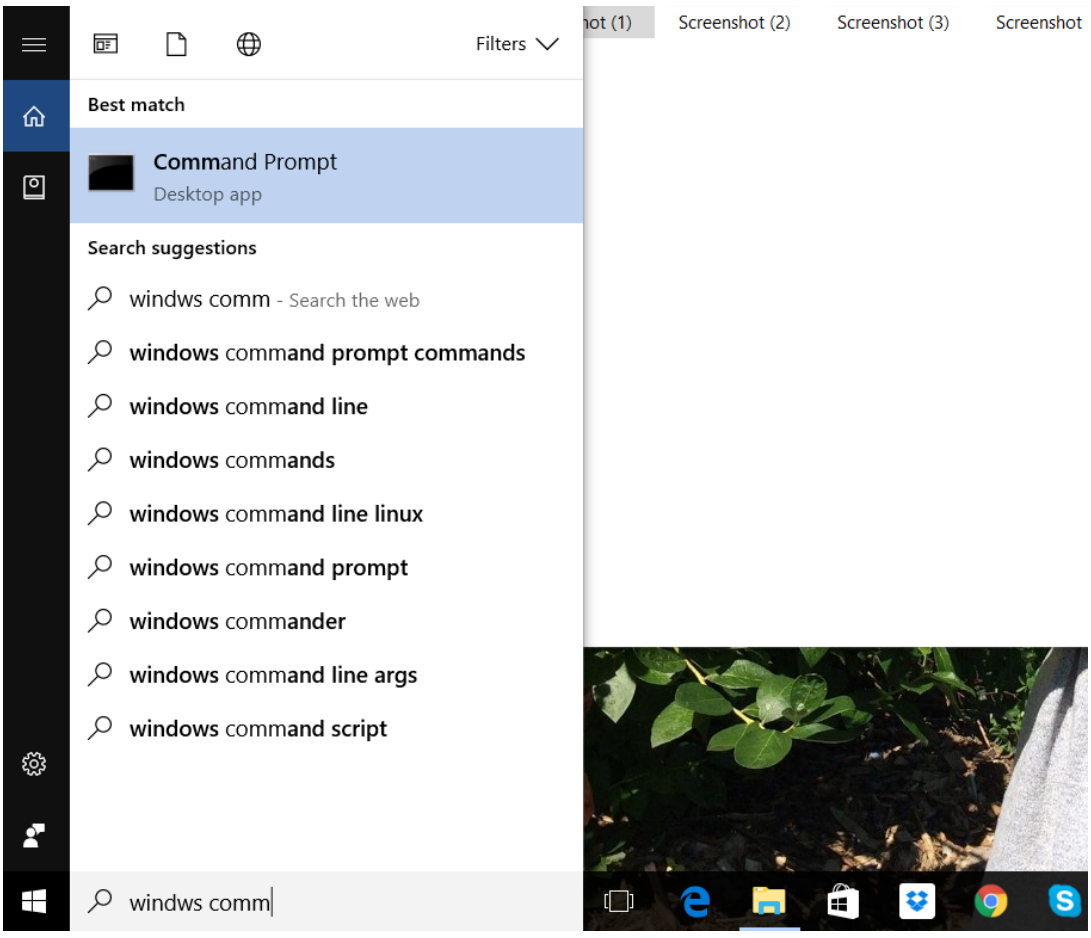


Bang! The setup is ready to install Node and NPM. Let's click on the Install button so hard!



Step 3: Check Node.js and NPM Version

- If you have a doubt whether you have installed everything correctly or not, let's verify it with "Command Prompt".



Command Prompt window will appear on the screen.

To confirm Node installation, type `node -v` command.

To confirm NPM installation, type `npm -v` command.

And you don't need to worry if you see different numbers than mine as Node and NPM are updated frequently.

In my case, the version of node.js is v14.15.3 and npm is 6.14.9.

We just seen that we have installed NPM with Node.Js so, let's now learn that what is NPM ?

NPM(Node Package Manager):

NPM – or "Node Package Manager" – is the default package manager for JavaScript's runtime Node.js.

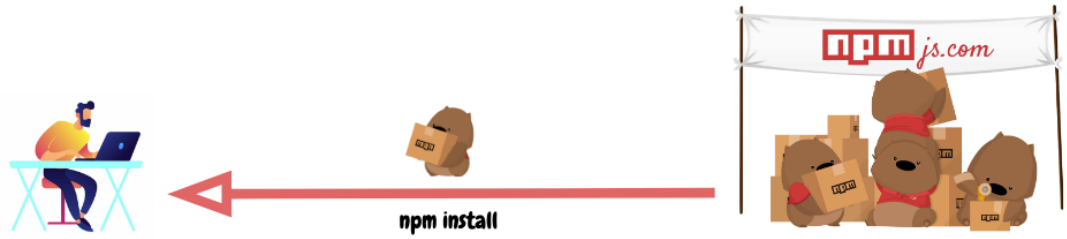
It's also known as "Ninja Pumpkin Mutants", "Nonprofit Pizza Makers", and a host of other random names that you can explore and probably contribute over at npm-expansions.

NPM consists of two main parts:

- a CLI (command-line interface) tool for publishing and downloading packages, and
- an online repository that hosts JavaScript packages

For a more visual explanation, we can think of the repository npmjs.com as a fulfillment center that receives packages of goods from sellers (np package authors) and distributes these goods to buyers (npm package users).

To facilitate this process, the npmjs.com fulfillment center employs an army of hardworking wombats (npm CLI) who will be assigned as person assistants to each individual npmjs.com customer. So dependencies are delivered to JavaScript developers like this:



and the process of publishing a package for your JS mates would be something like this:

Let's look at how this army of wombats assist developers who want to use JavaScript packages in their projects. We'll also see how they help ope source wizards get their cool libraries out into the world.

package.json

Every project in JavaScript – whether it's Node.js or a browser application – can be scoped as an npm package with its own package information ar its `package.json` job to describe the project.

We can think of `package.json` as stamped labels on those npm good boxes that our army of Wombats delivers around.

`package.json` will be generated when `npm init` is run to initialise a JavaScript/Node.js project, with these basic metadata provided by developers

- `name` : the name of your JavaScript library/project
- `version` : the version of your project. Often times, for application development, this field is often neglected as there's no apparent need for versioning opensource libraies. But still, it can come handy as a source of the deployment's version.
- `description` : the project's description
- `license` : the project's license

Downloads

`npm` manages downloads of dependencies of your project.

Installing all dependencies

If a project has a `package.json` file, by running

```
npm install
```

it will install everything the project needs, in the `node_modules` folder, creating it if it's not existing already.

Installing a single package

You can also install a specific package by running

```
npm install <package-name>
```

Furthermore, since npm 5, this command adds `<package-name>` to the `package.json` file *dependencies*. Before version 5, you needed to add th flag `--save` .

Often you'll see more flags added to this command:

- `-save-dev` installs and adds the entry to the `package.json` file *devDependencies*
- `-no-save` installs but does not add the entry to the `package.json` file *dependencies*
- `-save-optional` installs and adds the entry to the `package.json` file *optionalDependencies*
- `-no-optional` will prevent optional dependencies from being installed

Shorthands of the flags can also be used:

- S: --save
- D: --save-dev
- O: --save-optional

The difference between *devDependencies* and *dependencies* is that the former contains development tools, like a testing library, while the latter bundled with the app in production.

YARN:

Facebook developed Yarn in 2016 as a replacement for NPM. It was designed to offer more advanced features that NPM lacked at the time (such a version locking) and create a more secure, stable, and efficient product.

However, since Yarn was released, NPM has added several crucial features. In its current state, Yarn is now more of an alternative to NPM rather than replacement.

Yarn version 1 and NPM both manage dependencies in a very similar way. They both store project metadata in the **package.json** file, located the *node_modules* folder inside the project directory.

Starting from version 2, Yarn no longer uses the *node_modules* folder to track dependencies. Instead, Yarn 2.0 uses the **Plug'n'Play** feature, whic generates a single **.pnp.cjs** file. This file contains a map of the dependency hierarchy for a project.

Yarn uses the `yarn` command to install dependencies. It installs dependencies in parallel, allowing you to add multiple files at the same time.

Installing dependencies automatically creates a lock file that saves the exact list of dependencies used for the project. With Yarn, this file called **yarn.lock**.

NPX: Node Package Runner

`npx` is a very powerful command that's been available in **npm** starting version 5.2, released in July 2017.

`npx` lets you run code built with Node.js and published through the npm registry.

Easily run local commands

Node.js developers used to publish most of the executable commands as global packages, in order for them to be in the path and executab immediately.

This was a pain because you could not really install different versions of the same command.

Running `npx commandname` automatically finds the correct reference of the command inside the `node_modules` folder of a project, without needir to know the exact path, and without requiring the package to be installed globally and in the user's path.

Installation-less command execution

There is another great feature of `npx` , which is allowing to run commands without first installing them.

This is pretty useful, mostly because:

1. you don't need to install anything
2. you can run different versions of the same command, using the syntax @version

A typical demonstration of using `npx` is through the `cowsay` command. `cowsay` will print a cow saying what you wrote in the command. F example:

`cowsay "Hello"` will print

```
 _______
< Hello >
 -------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
```

This only works if you have the `cowsay` command globally installed from npm previously. Otherwise you'll get an error when you try to run th command.

`npx` allows you to run that npm command without installing it first. If the command isn't found, `npx` will install it into a central cache:

```
npx cowsay "Hello"
```

will do the job.

Now, this is a funny useless command. Other scenarios include:

- running the `vue` CLI tool to create new applications and run them: `npx @vue/cli create my-vue-app`
- creating a new React app using `create-react-app` : `npx create-react-app my-react-app`

and many more.

Now, let's create a new node project:

Creation of a Node project:

First, let's create our project directory, then we run the init command

```
mkdir myapp
cd myapp
npm init
```

Follow the command line prompt and you can leave everything as the default, then it will ask you in the end if everything looks ok.. click yes. Yo directory should look something like this

```
|_myapp
|__package.json
```

The *package.json* file consists of all the project settings and other npm package dependencies. If you open the *package.json* it should look like this

```
{
"name": "myapp",
"version": "1.0.0",
"description": "",
"main": "index.js",
"scripts": {"test": "echo \"Error: no test specified\" && exit 1"},
"author": "",
"license": "ISC"
}
```

The "name" element is the name of your package and if you upload your package to the npm registry it will be under the string that you set as the valu of this element, in our case it's "myapp".

Now, create a file named `index.js`

Now we can safely run our project. It is as simple as running the following command

```
node index.js
```

Modules in Node.Js

In Node.js, **Modules** are the blocks of encapsulated code that communicates with an external application on the basis of their related functionalit Modules can be a single file or a collection of multiples files/folders. The reason programmers are heavily reliant on modules is because of their r usability as well as the ability to break down a complex piece of code into manageable chunks.

Modules are encapsulated code blocks that communicate with an external application. These can be a single file or a collection of multiple files/folder These are reusable, hence they are widely used.

Types of Modules in Node.Js

There are three types of modules

1. Core Modules
2. local Modules
3. Third-party Modules

1. Nodejs Core Modules:

Built-in modules of node.js that are part of nodejs and come with the Node.js installation process are known as core modules.

To load/include this module in our program, we use the **require** function.

```
let  module = require('module_name')
```

The return type of require() function depends on what the particular module returns.

Http, file system and url modules are some of the core modules.

2. Nodejs Local Modules:

Local modules are created by us locally in our Node.js application. These modules are included in our program in the same way as we include the built module.

Let's build a module with the name as sum to add two numbers and include them in our index.js file to use them.

Code for creating local modules and exporting:

```
exports.add=function(n,m){
    return n+m;
};
```

Exports keyword is used to make properties and methods available outside the file.

In order to include the add function in our index.js file we use the require function.

Code for including local modules:

```
let sum = require('./sum')

console.log("Sum of 10 and 20 is ", sum.add(10, 20))
```

Add the above code in a index.js file

To run this file, open a terminal in the project directory and type node index.js and press enter. You can see the result of addition of 10 and 20. Th addition has been performed by the add function in the sum module.

3. Nodejs Third Party Modules:

Modules that are available online and are installed using the npm are called third party modules. Examples of third party modules are expres mongoose, etc.

Now, after seeing what are modules and their types let's learn in detail about the main modules used in Node.Js.

HTTP Module:

The `http` module is a core module of Node designed to support many features of the HTTP protocol.

The following example shows how to use the `http` module:

First, create a new file called `server.js` and include the `http` module by using the `require()` function:

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

Second, create an HTTP server using the `createServer()` method of the `http` object.

```
const server = http.createServer((req, res) => {
    if (req.url === '/') {
        res.write('<h1>Hello, Node.js!</h1>');
    }
    res.end();
});
```

The `createServer()` accepts a callback that has two parameters: HTTP request (`req`) and response (`res`). Inside the callback, we send a HTML string to the browser if the URL is `/` and end the request.

Third, listen to the incoming HTTP request on the port `5000` :

```
server.listen(5000);
console.log(`The HTTP Server is running on port 5000`);
```

Put it all together:

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

const server = http.createServer((req, res) => {
    if (req.url === '/') {
        res.write('<h1>Hello, Node.js!</h1>');
    }
    res.end();
});

server.listen(5000);
console.log(`The HTTP Server is running on port 5000`);
```

The following starts the HTTP server:

```
node server.js
```

Output:

```
The HTTP Server is running on port 5000
```

Now, you can launch the web browser and go to the URL http://localhost:5000/. You'll see the following message:

```
Hello, Node.js
```

This simple example illustrates how to use the `http` module. In practice, you will not use the `http` module directly.

File System Module:

Node.js includes **fs** module to access physical file system. The fs module is responsible for all the asynchronous or synchronous file I/O operations.

Let's see some of the common I/O operation examples using fs module.

Reading File

Use fs.readFile() method to read the physical file asynchronously.

Syntax:

```
fs.readFile(fileName [,options], callback)
```

Parameter Description:

- filename: Full path and name of the file as a string.
- options: The options parameter can be an object or string which can include encoding and flag. The default encoding is utf8 and default flag is "r".
- callback: A function with two parameters err and fd. This will get called when readFile operation completes.

The following example demonstrates reading existing TestFile.txt asynchronously.

```
var fs = require('fs');

fs.readFile('TestFile.txt', function (err, data) {
                    if (err) throw err;

    console.log(data);
});
```

The above example reads TestFile.txt (on Windows) asynchronously and executes callback function when read operation completes. This read operatic either throws an error or completes successfully. The err parameter contains error information if any. The data parameter contains the content of th specified file.

The following is a sample `TextFile.txt` file.

```
This is test file to test fs module of Node.js
```

Now, run the above example and see the result as shown below.

```
C:\> node server.js
This is test file to test fs module of Node.js
```

Use `fs.readFileSync()` method to read file synchronously as shown below.

```
var fs = require('fs');

var data = fs.readFileSync('dummyfile.txt', 'utf8');
console.log(data);
```

Writing File

Use fs.writeFile() method to write data to a file. If file already exists then it overwrites the existing content otherwise it creates a new file and writes da into it.

Signature:

```
fs.writeFile(filename, data[, options], callback)
```

Parameter Description:

- filename: Full path and name of the file as a string.
- Data: The content to be written in a file.
- options: The options parameter can be an object or string which can include encoding, mode and flag. The default encoding is utf8 and default flag is "r".
- callback: A function with two parameters err and fd. This will get called when write operation completes.

The following example creates a new file called test.txt and writes "Hello World" into it asynchronously.

```
var fs = require('fs');

fs.writeFile('test.txt', 'Hello World!', function (err) {
                        if (err)
        console.log(err);
```

```
                        else
        console.log('Write operation complete.');
});
```

In the same way, use `fs.appendFile()` method to append the content to an existing file.

```
var fs = require('fs');

fs.appendFile('test.txt', 'Hello World!', function (err) {
                        if (err)
        console.log(err);
                        else
        console.log('Append operation complete.');
});
```

Open File

Alternatively, you can open a file for reading or writing using fs.open() method.

Syntax:

```
fs.open(path, flags[, mode], callback)
```

Parameter Description:

- path: Full path with name of the file as a string.
- Flag: The flag to perform operation
- Mode: The mode for read, write or readwrite. Defaults to 0666 readwrite.
- callback: A function with two parameters err and fd. This will get called when file open operation completes.

Path Module:

The Node.Js Path module is a built-in module that helps you work with file system paths in an OS-independent way. The path module is essential you're building a CLI tool that supports OSX, Linux, and Windows.

Joining path modules in Node

The most commonly used function in the path module is `path.join()` . The `path.join()` function merges one or more path segments into a sing string, as shown below.

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

path.join('/path', 'to', 'test.txt'); // '/path/to/test.txt'
```

You may be wondering why you'd use the `path.join()` function instead of using string concatenation.

```
'/path' + '/' + 'to' + '/' + 'test.txt'; // '/path/to/test.txt'

['/path', 'to', 'test.txt'].join('/'); // '/path/to/test.txt'
```

There are two main reasons why.

First, for Windows support. Windows uses backslashes (`\`) rather than forward slashes (`/`) as path separators. The `path.join()` function handle this for you because `path.join('data', 'test.txt')` returns `'data/test.txt'` on both Linux and OSX, and `'data\\test.txt'` Windows.

Secondly, for handling edge cases. Numerous edge cases pop up when working with file system paths. For example, you may accidentally end up with duplicate path separator if you try to join two paths manually. The `path.join()` function handles leading and trailing slashes for you, like so:

```
path.join('data', 'test.txt'); // 'data/test.txt'
path.join('data', '/test.txt'); // 'data/test.txt'
path.join('data/', 'test.txt'); // 'data/test.txt'
path.join('data/', '/test.txt'); // 'data/test.txt'
```

Parsing paths in Node

The path module also has several functions for extracting path components, such as the file extension or directory. For exampl the `path.extname()` function returns the file extension as a string:

```
path.extname('/path/to/test.txt'); // '.txt'
```

Like joining two paths, getting the file extension is trickier than it first seems. Taking everything after the last `.` in the string doesn't work if there's directory with a `.` in the name, or if the path is a dotfile.

```
path.extname('/path/to/github.com/README'); // ''

path.extname('/path/to/.gitignore'); // ''
```

The path module also has `path.basename()` and `path.dirname()` functions, which get the file name (including the extension) and director respectively.

```
path.basename('/path/to/test.txt'); // 'test.txt'

path.dirname('/path/to/test.txt'); // '/path/to'
```

Do you need both the extension and the directory? The `path.parse()` function returns an object containing the path broken up into five differe components, including the extension and directory. The `path.parse()` function is also how you can get the file's name without any extension.

```
/*
{
  root: '/',
  dir: '/path/to',
  base: 'test.txt',
  ext: '.txt',
  name: 'test'
}
*/
path.parse('/path/to/test.txt');
```

URL Module:

The URL module splits up a web address into readable parts.

To include the URL module, use the `require()` method:

```
var url = require('url');
```

Parse an address with the `url.parse()` method, and it will return a URL object with each part of the address as properties:

In the example below, we are splitting a web url:

```
var url = require('url');
var adr = 'http://localhost:8080/default.htm?year=2017&month=february';
var q = url.parse(adr, true);

console.log(q.host); //returns 'localhost:8080'
console.log(q.pathname); //returns '/default.htm'
console.log(q.search); //returns '?year=2017&month=february'

var qdata = q.query; //returns an object: { year: 2017, month: 'february' }
console.log(qdata.month); //returns 'february'
```

NOTE:

Using the URL module will not let you read the URL you receive from the request. This module let you parse a URL string. If you want to read the UF you recieve from a request in Node.Js, you have to use the HTTP module in Node.Js.

OS Module:

To use the `os` module, you include it as follows:

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

The `os` module provides you with many useful properties and methods for interacting with the operating system and server.

For example, the `os.EOL` property returns the platform-specific end-of-line marker.

The `os.EOL` property returns `\r\n` on Windows and `\n` on Linux or macOS.

Getting the current Operating System information

The `os` module provides you with some useful methods to retrieve the operating system of the server. For example:

```
let currentOS = {
    name: os.type(),
    architecture: os.arch(),
    platform: os.platform(),
    release: os.release(),
    version: os.version()
};

console.log(currentOS);
```

Output:

```
{
    name: 'Windows_NT',
    architecture: 'x64',
    platform: 'win32',
    release: '10.0.18362',
    version: 'Windows 10 Pro'
}
```

Checking server uptime

The `os.uptime()` method returns the system uptime in seconds. For example:

```
console.log(`The server has been up for ${os.uptime()} seconds.`);
```

Output:

```
The server has been up for 44203 seconds.
```

Getting the current user information

The `os.userInfo()` method returns the information about the current user:

```
console.log(os.userInfo());
```

Output:

```
{
    uid: -1,
    gid: -1,
    username: 'john',
    homedir: 'C:\\Users\\john',
    shell: null
}
```

Getting the server hardware information

The `os.totalmem()` method returns the total memory in bytes of the server:

```
let totalMem = os.totalmem();
console.log(totalMem);
```

Output:

```
8464977920
```

To get the amount of free memory in bytes, you use the `os.freemem()` method:

```
let freeMem = os.freemem();
console.log(freeMem);
```

Output:

```
1535258624
```

Node.Js argv property:

The **process.argv property** is an inbuilt application programming interface of the process module which is used to get the arguments passed to th node.js process when run in the command line.

Syntax:

```
process.argv
```

Return Value: This property returns an array containing the arguments passed to the process when run it in the command line. The first element is th process execution path and the second element is the path for the js file.

Below examples illustrate the use of **process.argv property** in Node.js:

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

// Printing process.argv property value
console.log(process.argv);
```

Output:

```
[ 'C:\\Program Files\\nodejs\\node.exe',
  'C:\\nodejs\\g\\process\\argv_1.js',
  'extra_argument1',
  'extra_argument2',
  '3'
]
```

So, these are the main modules in Node.js and we will learn about some other modules in the next session.

Conclusion:

In this session, we have learnt about:

- What is Node.Js, multi-page applications
- Installation of Node.Js and NPM, yarn and NPX
- What are the modules in Node.Js and about some main modules.

Interview Questions:

> What is Node.Js and Where you can use it ?

Node.Js is an open source, cross-platform Javascript runtime environment and library to run web applications outside the client's browser**.** It is use to create server-side web applications.

Node.js is perfect for data-intensive applications as it uses an asynchronous, event-driven model. You can use I/O intensive web applications like vide streaming sites. You can also use it for developing: Real-time web applications, Network applications, General-purpose applications, and Distribute systems.

> What is NPM ?

NPM stands for Node Package Manager, responsible for managing all the packages and modules for Node.js.

Node Package Manager provides two main functionalities:

- Provides online repositories for node.js packages/modules, which are searchable on search.nodejs.org
- Provides command-line utility to install Node.js packages and also manages Node.js versions and dependencies

> What are modules in Node.JS ? Name any three main modules in Node.Js.

Modules are like Javascript libraries that can be used in a Node.js application to include a set of functions. To include a module in a Node.js applicatio use the **require()** function with the parentheses containing the module's name.

Node.js has many modules to provide the basic functionality needed for a web application. Some of them include:

Core Modules	Description
HTTP	Includes classes, methods, and events to create a Node.js HTTP server
fs	Includes events, classes, and methods to deal with file I/O operations
url	Includes methods for URL parsing

Thank You !