ETHNUS™

Explore | Expand | Enrich

15 YEARS

<Codemithra />™

Introduction

- Node.js is a very powerful JavaScript-based platform built on Google Chrome's JavaScript V8 Engine.
- It is used to develop I/O intensive web applications like video streaming sites, single-page applications, and other web applications.
- Node.js is open source, completely free, and used by thousands of developers around the world.

What is NodeJS?

•Node.js is a server-side platform built on Google Chrome's JavaScript Engine (V8 Engine). Node.js was developed by Ryan Dahl in 2009 and its latest version is v0.10.36. The definition of Node.js as supplied by its official documentation is as follows –

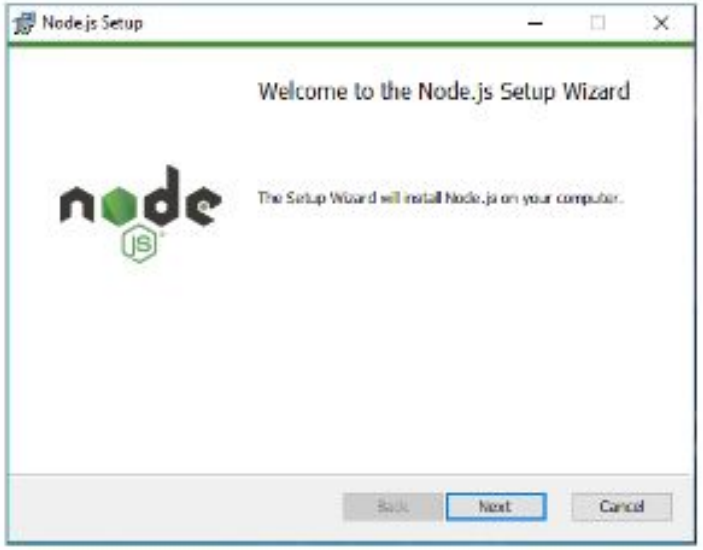
•Node.js is a platform built on Chrome's JavaScript runtime for easily building fast and scalable network applications.It uses an event-driven, non-blocking I/O model that makes it lightweight and efficient, perfect for data-intensive real-time applications that run across distributed devices.

•Node.js also provides a rich library of various JavaScript modules which simplifies the development of web applications using Node.js to a great extent.

•Node.js = Runtime Environment + JavaScript Library

•

Install Node js on Windows

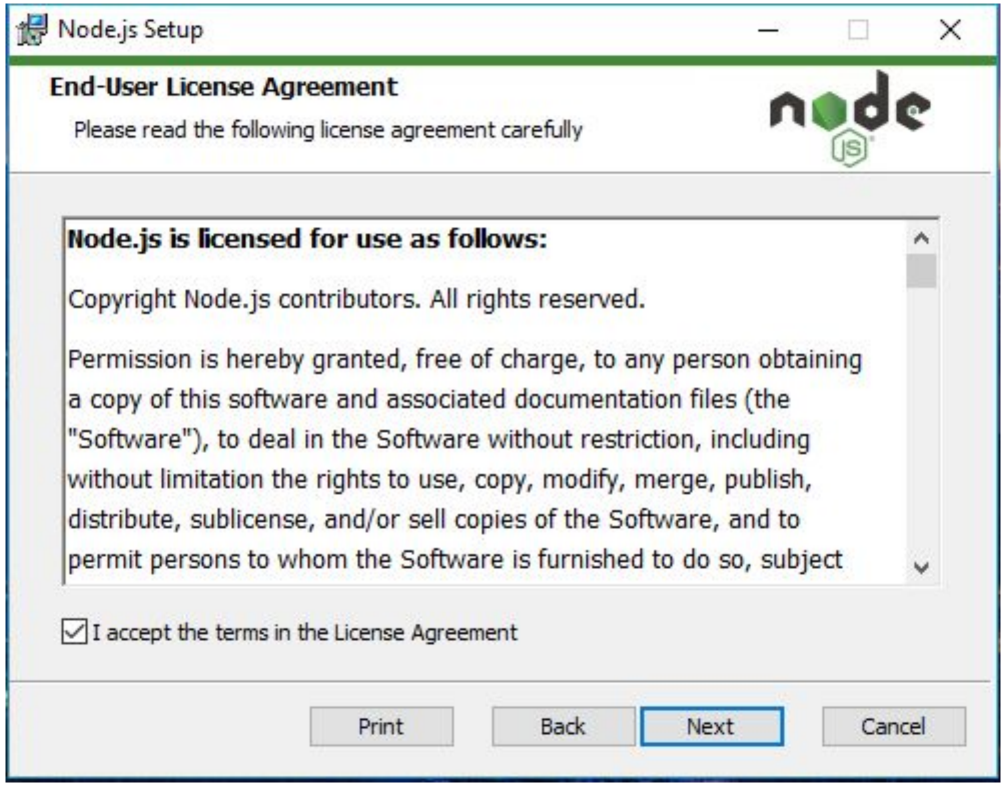


Node.js Setup

Welcome to the Node.js Setup Wizard

The Setup Wizard will install Node.js on your computer.

After clicking “Next”, End-User License Agreement (EULA) will open.Check “I accept the terms in the License Agreement”.Select “Next”.



Node.js Setup

End-User License Agreement

Please read the following license agreement carefully

Node.js is licensed for use as follows:

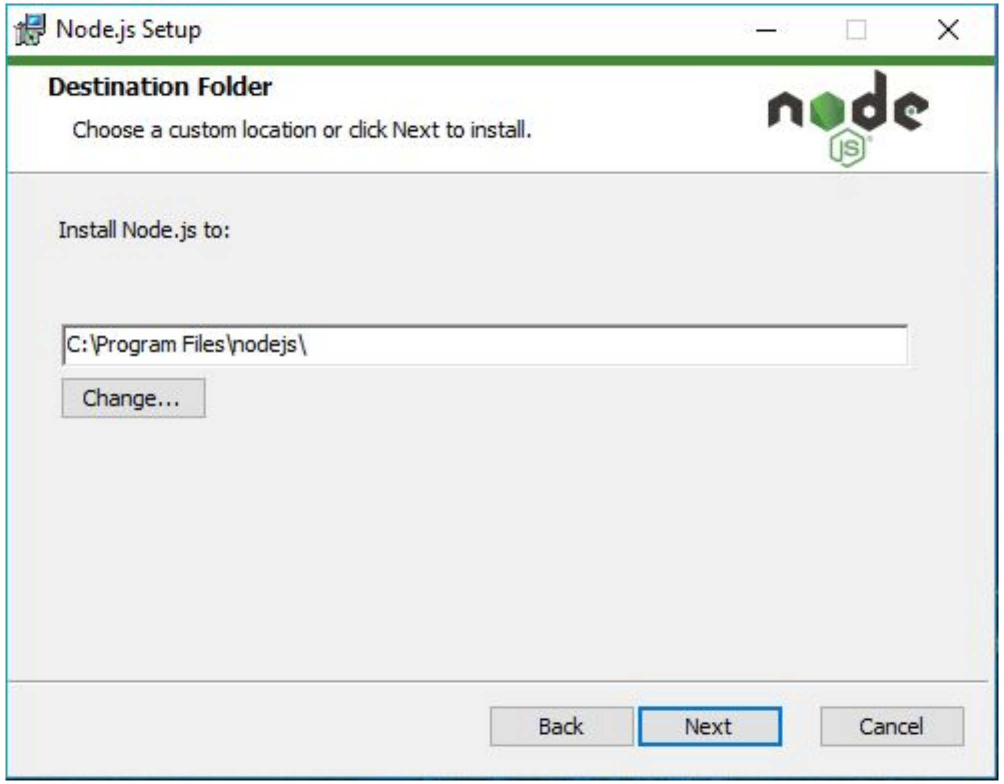
Copyright Node.js contributors. All rights reserved.

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject

☑ I accept the terms in the License Agreement

Setup

- Set the Destination Folder where you want to install Node.js & Select “Next”.



Node.js Setup

Destination Folder

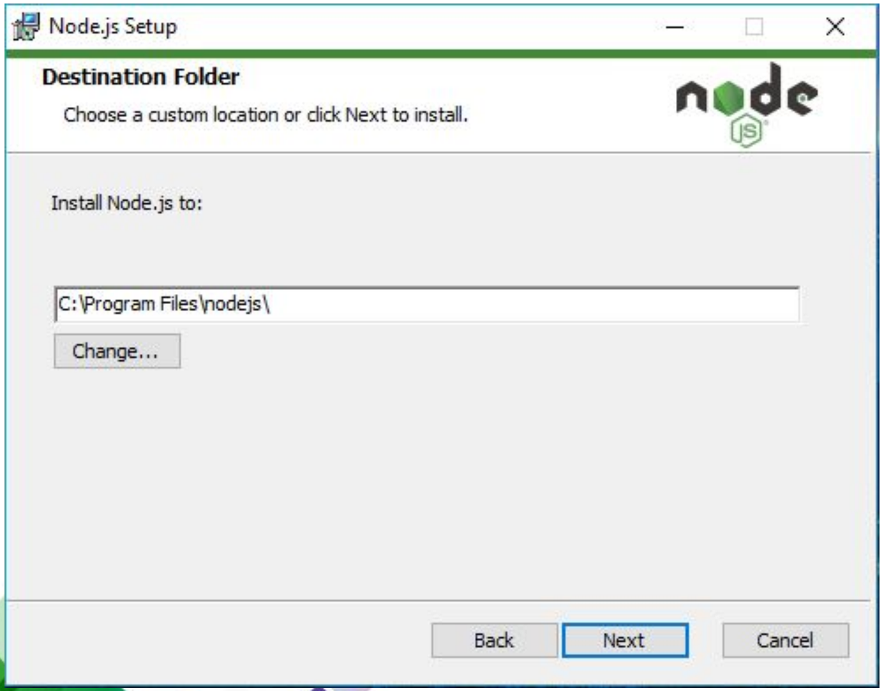
Choose a custom location or click Next to install.

Install Node.js to:

C:\Program Files\nodejs\

Setup

- Set the Destination Folder where you want to install Node.js & Select “Next”.



Node.js Setup

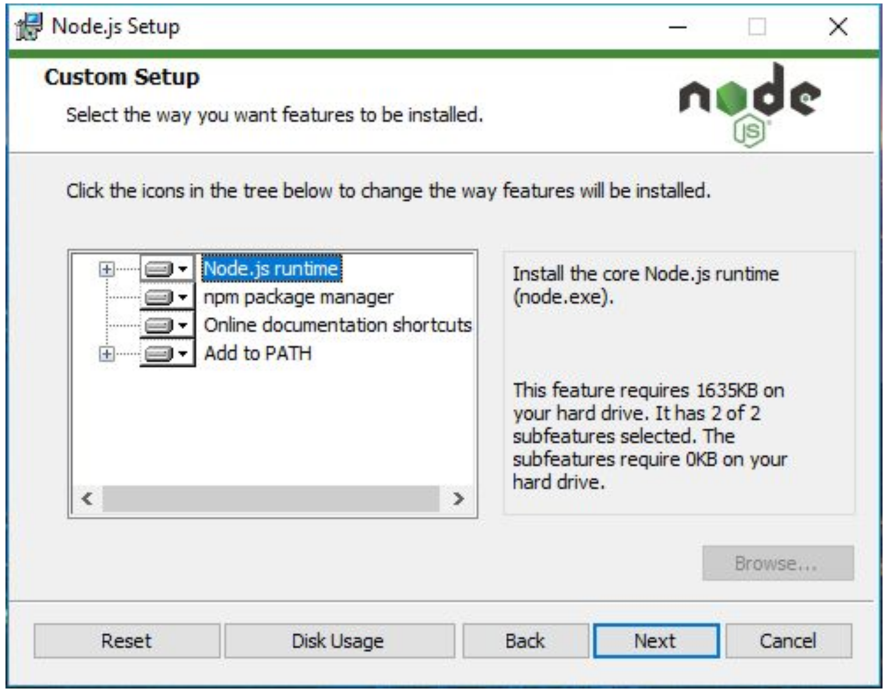
Destination Folder

Choose a custom location or click Next to install.

Install Node.js to:

C:\Program Files\nodejs\

Select “Next”.



Node.js Setup

Custom Setup

Select the way you want features to be installed.

Click the icons in the tree below to change the way features will be installed.

- Node.js runtime
- npm package manager
- Online documentation shortcuts
- Add to PATH

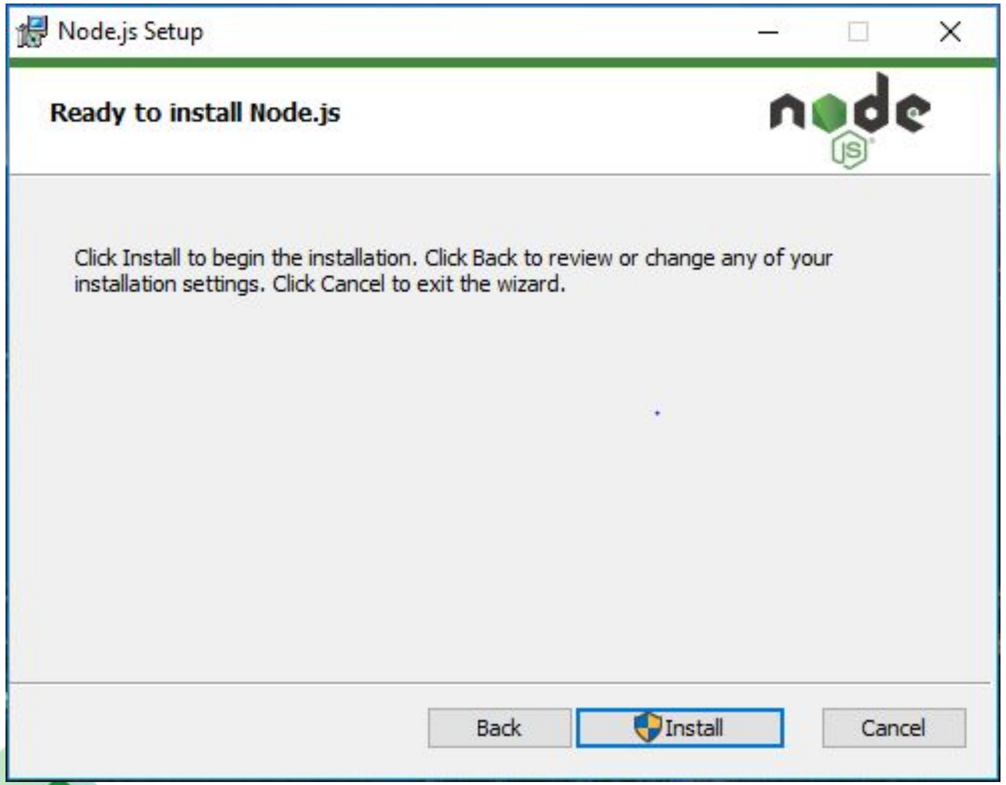
Install the core Node.js runtime (node.exe).

This feature requires 1635KB on your hard drive. It has 2 of 2 subfeatures selected. The subfeatures require 0KB on your hard drive.

`Setup

- Ready to Install Node.js.

  Select “Install”.

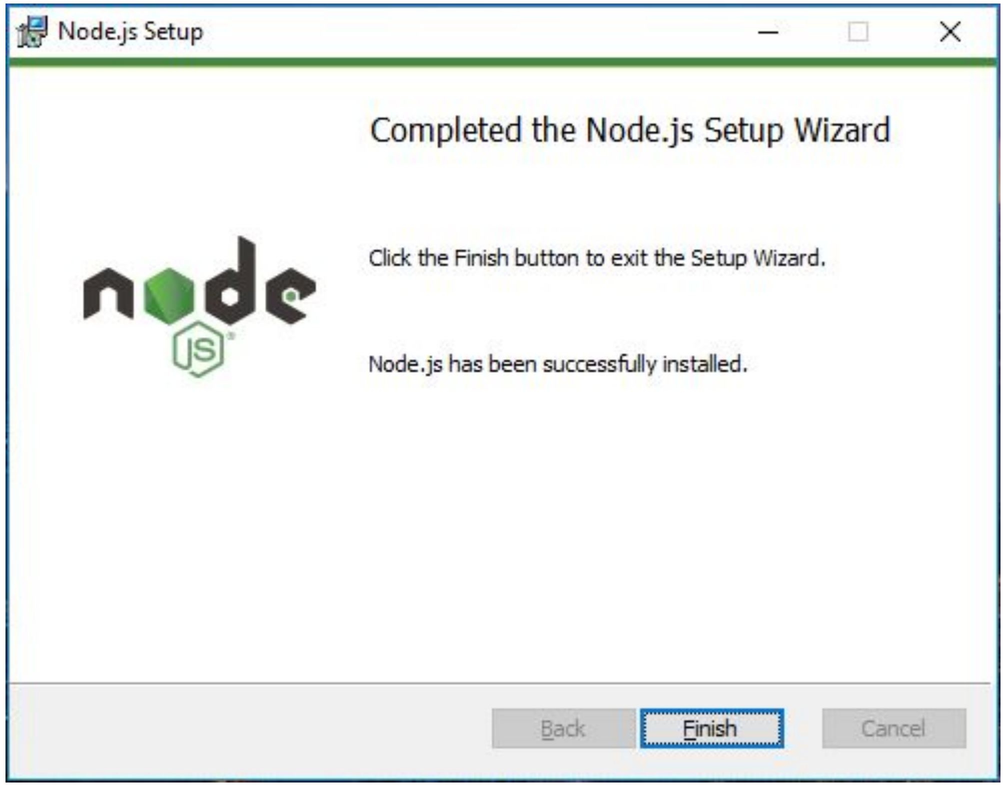


Node.js Setup

Ready to install Node.js

Click Install to begin the installation. Click Back to review or change any of your installation settings. Click Cancel to exit the wizard.

A prompt saying – “This step requires administrative privileges” will appear.Authenticate the prompt as an “Administrator”.Installing Node.js.Do not close or cancel the installer until the install is completeComplete the Node.js Setup Wizard.Click “Finish”.To check whether node js is installed properly type node -v in Command Prompt.



Node.js Setup

Completed the Node.js Setup Wizard

Click the Finish button to exit the Setup Wizard.

Node.js has been successfully installed.

Simple Node js Application

A Node.js application consists of the following three important components –

Import required modules – We use the require directive to load Node.js modules.

- Create server – A server which will listen to client's requests similar to Apache HTTP Server.
- Read request and return response – The server created in an earlier step will read the HTTP request made by the client which can be a browser or a console and return the response.

Step 1 - Import Required Module

We use the require directive to load the http module and store the returned HTTP instance into an http variable as follows –

```
var http = require("http");
```

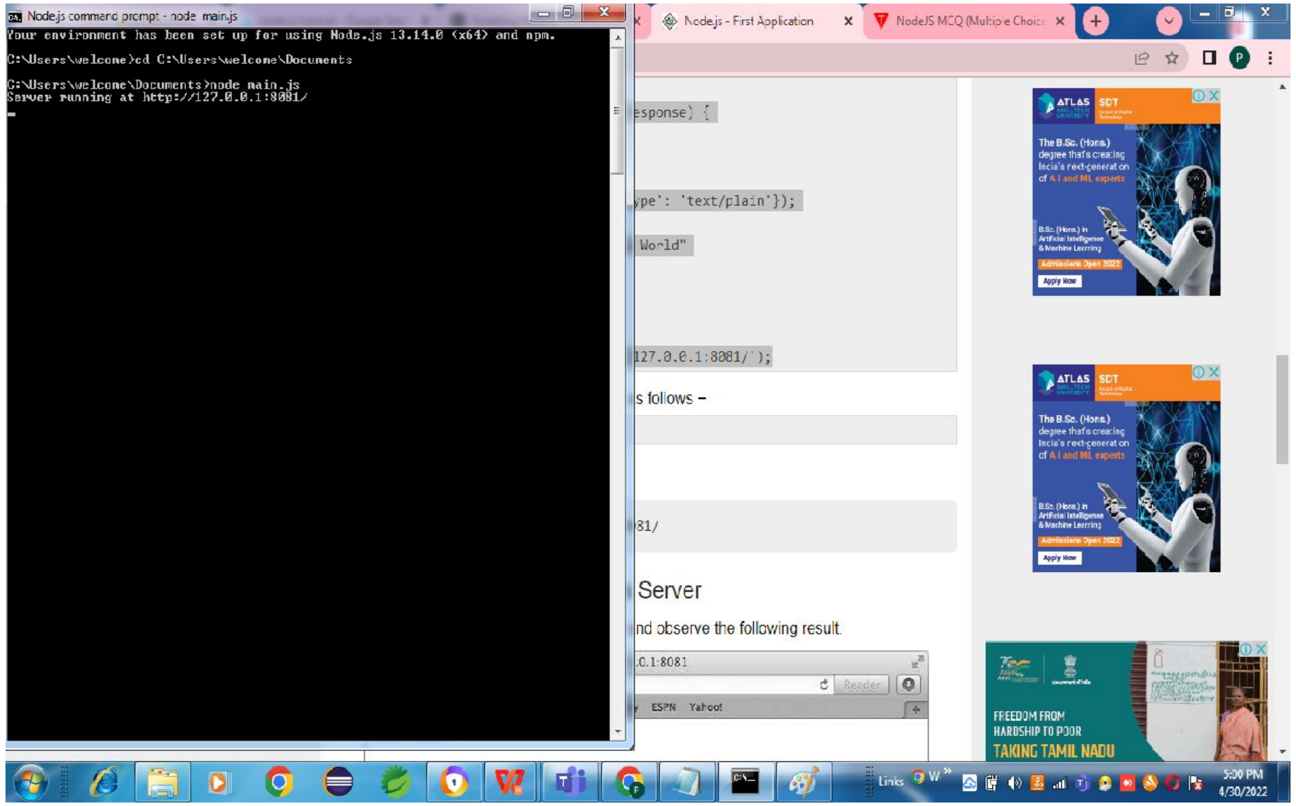
Simple Node js Application

Step 2 - Create Server
We use the created http instance and call http.createServer() method to create a server instance and then we bind it at port 8081 using the listen method associated with the server instance.
Pass it a function with parameters request and response. Write the sample implementation to always return "Hello World".

```
http.createServer(function (request, response) {
  // Send the HTTP header
  // HTTP Status: 200 : OK
  // Content Type: text/plain
  response.writeHead(200, {'Content-Type': 'text/plain'});

  // Send the response body as "Hello World"
  response.end('Hello World\n');
}).listen(8081);

// Console will print the message
console.log('Server running at http://127.0.0.1:8081/');
```

Output

Perform the following steps:

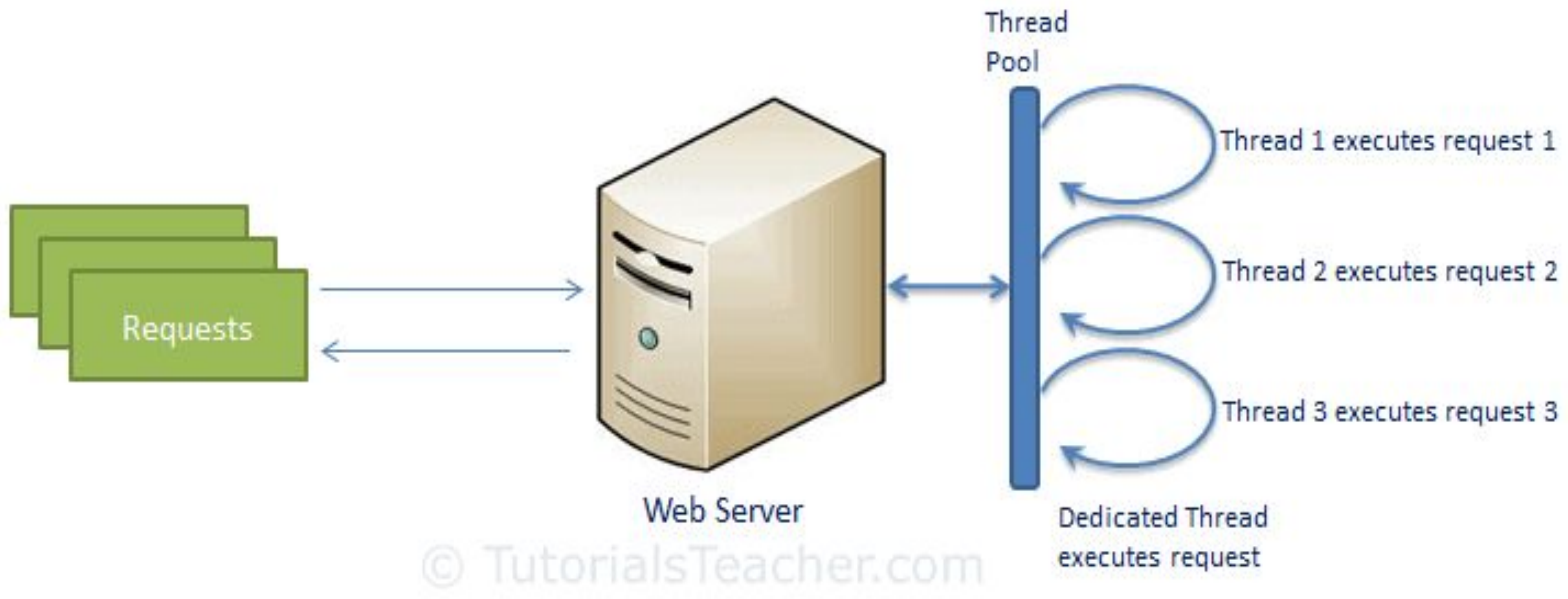

Node.js command prompt - node main.js
Node.js - First Application
NodeJS MCQ (Multiple Choice

```
Your environment has been set up for using Node.js 13.14.0 (x64) and npm.

C:\Users\welcome>cd C:\Users\welcome\Documents

C:\Users\welcome\Documents>node main.js
Server running at http://127.0.0.1:8081/
```

Traditional Web Server Model

In the traditional web server model, each request is handled by a dedicated thread from the thread pool. If no thread is available in the thread pool at any point of time then the request waits till the next available thread. Dedicated thread executes a particular request and does not return to thread pool until it completes the execution and returns a response.

Diagram


Thread Pool
Requests
Web Server
Thread 1 executes request 1
Thread 2 executes request 2
Thread 3 executes request 3
Dedicated Thread executes request
© TutorialsTeacher.com

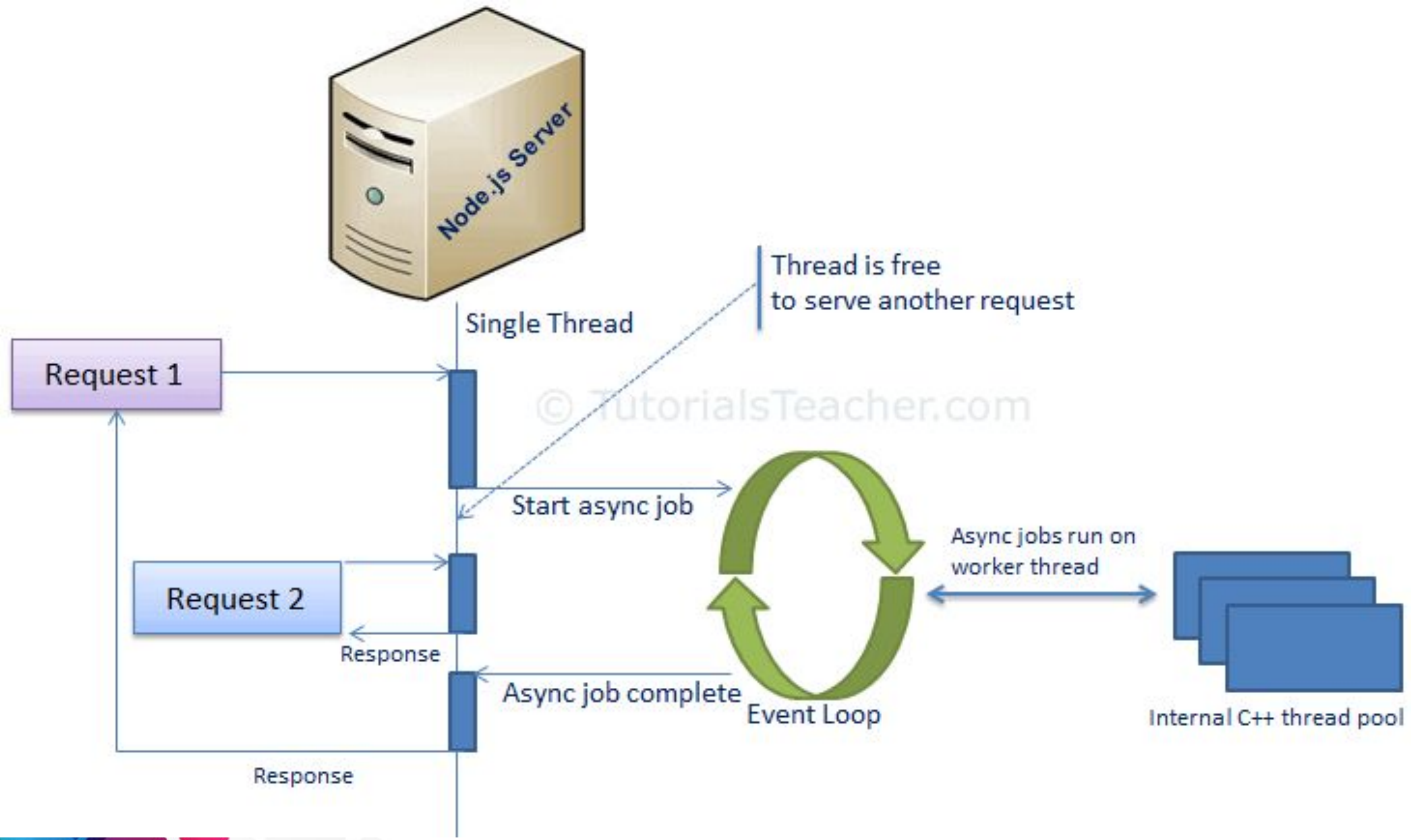
Node.js Process Model

- Node.js processes user requests differently when compared to a traditional web server model.
- Node.js runs in a single process and the application code runs in a single thread and thereby needs less resources than other platforms.
- All the user requests to your web application will be handled by a single thread and all the I/O work or long running job is performed asynchronously for a particular request.
- So, this single thread doesn't have to wait for the request to complete and is free to handle the next request. When asynchronous I/O work completes then it processes the request further and sends the response.

Node.js Process Model

- An event loop is constantly watching for the events to be raised for an asynchronous job and executing callback function when the job completes.
- Internally, Node.js uses libev for the event loop which in turn uses internal C++ thread pool to provide asynchronous I/O.

Diagram



Node.js Server
Single Thread
Thread is free to serve another request
Request 1
© TutorialsTeacher.com
Start async job
Async jobs run on worker thread
Request 2
Response
Async job complete
Event Loop
Internal C++ thread pool
Response

REPL Terminal

- REPL stands for Read Eval Print Loop and it represents a computer environment like a Windows console or Unix/Linux shell where a command is entered and the system responds with an output in an interactive mode. Node.js or Node comes bundled with a REPL environment. It performs the following tasks – **Read** – Reads user's input, parses the input into JavaScript data-structure, and stores in memory.
- **Eval** – Takes and evaluates the data structure.
- **Print** – Prints the result.
- **Loop** – Loops the above command until the user presses ctrl-c twice.
- The REPL feature of Node is very useful in experimenting with Node.js codes and to debug JavaScript codes

Event Loop

Node.js is a single-threaded application, but it can support concurrency via the concept of event and callbacks.

Every API of Node.js is asynchronous and being single-threaded, they use async function calls to maintain concurrency.

Node uses observer pattern.

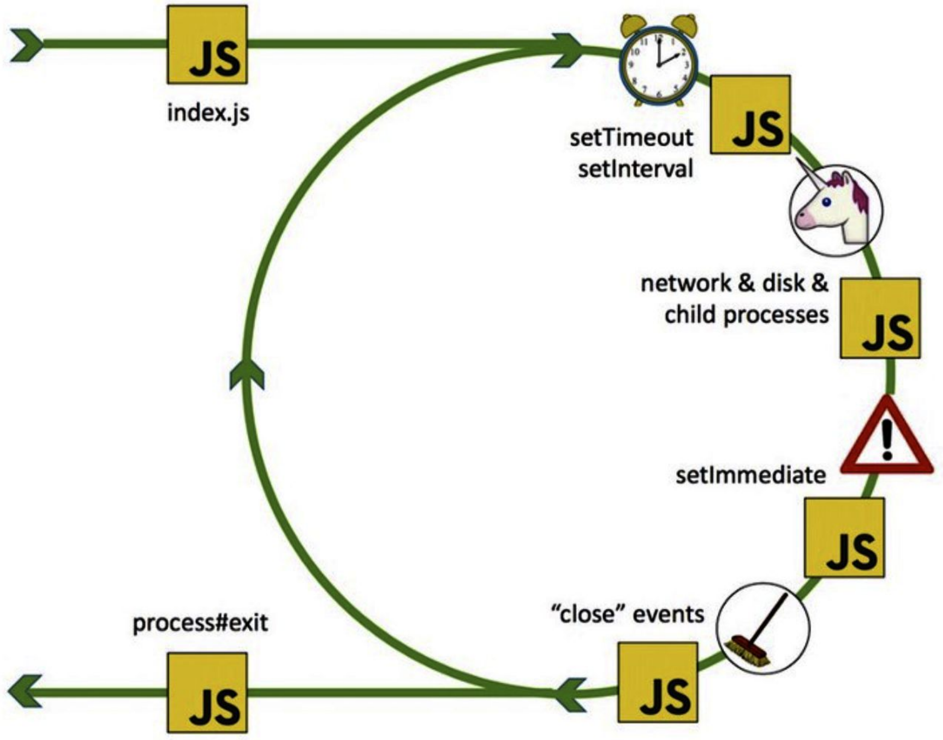
Node thread keeps an event loop and whenever a task gets completed, it fires the corresponding event which signals the event-listener function to execute.

Event Driven Programming

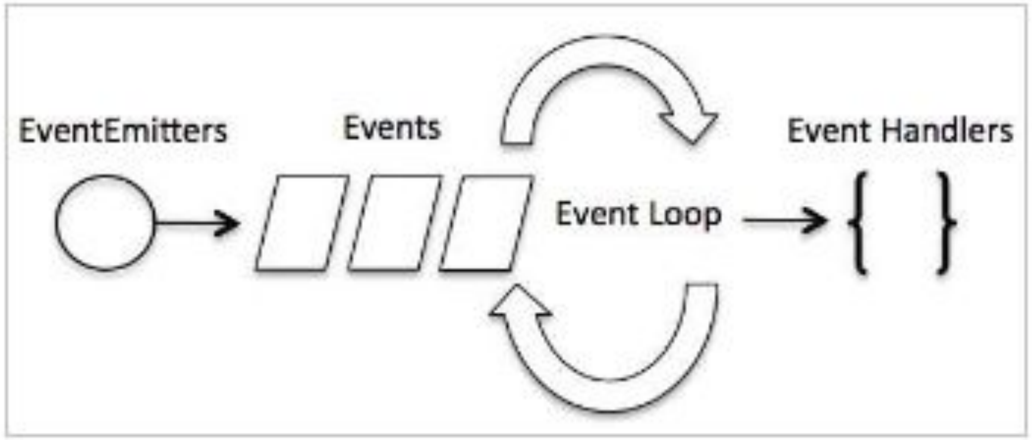
- Node.js uses events heavily and it is also one of the reasons why Node.js is pretty fast compared to other similar technologies.
- As soon as Node starts its server, it simply initiates its variables, declares functions and then simply waits for the event to occur.
- In an event-driven application, there is generally a main loop that listens for events, and then triggers a callback function when one of those events is detected.
- Although events look quite similar to callbacks, the difference lies in the fact that callback functions are called when an asynchronous function returns its result, whereas event handling works on the observer pattern.
- The functions that listen to events act as Observers.
- Whenever an event gets fired, its listener function starts executing. Node.js has multiple in-built events available through events module and EventEmitter class which are used to bind events and event-listeners as follows –

Event driven programming

```
// Import events module
var events = require('events');

// Create an eventEmitter object
var eventEmitter = new events.EventEmitter();
Following is the syntax to bind an event handler with an event −

// Bind event and event  handler as follows
eventEmitter.on('eventName', eventHandler);
We can fire an event programmatically as follows −

// Fire an event
eventEmitter.emit('eventName');
```



Diagram



Example

```
var events = require('events’); // Import events module
var eventEmitter = new events.EventEmitter(); // Create an eventEmitter object
var connectHandler = function connected() {// Create an event handler as follows
   console.log('connection successful.’);
   eventEmitter.emit('data_received'); // Fire the data_received event
}
// Bind the connection event with the handler
eventEmitter.on('connection', connectHandler);
 // Bind the data_received event with the anonymous function
eventEmitter.on('data_received', function() {
   console.log('data received successfully.');
});
// Fire the connection event
eventEmitter.emit('connection');
console.log("Program Ended");
```

output

```
C:\Users\welcome\Documents>node main.js
The output will appear as:
Connection successful
Data received successfully
Program Ended
```

Callbacks Concept

- **What is Callback?**
- Callback is an asynchronous equivalent for a function.
- A callback function is called at the completion of a given task.
- Node makes heavy use of callbacks.
- All the APIs of Node are written in such a way that they support callbacks.
- For example, a function to read a file may start reading file and return the control to the execution environment immediately so that the next instruction can be executed.
- Once file I/O is complete, it will call the callback function while passing the callback function, the content of the file as a parameter. So there is no blocking or wait for File I/O.
- This makes Node.js highly scalable, as it can process a high number of requests without waiting for any function to return results.

Example

Create a file input.txt with the following content:

```
 Codemithra is giving self learning content on AWS,Aptitude
and mern stack
to teach the world in simple and easy way!!!!!
```

Create another file main.js and add the following code:

```
var fs = require("fs");
var data = fs.readFileSync('input.txt');

console.log(data.toString());
console.log("Program Ended");
```

Now compile the main.js file and see the following result:

```
 Codemithra is giving self learning content on AWS,Aptitude
and mern stack
to teach the world in simple and easy way!!!!!
Program Ended
```

Blocking & Non Blocking Code

Blocking refers to operations that block further execution until that operation finishes while **non-blocking** refers to code that doesn't block execution.

Or as Node.js docs puts it, blocking is when the execution of additional JavaScript in the Node.js process must wait until a non-JavaScript operation completes.

Blocking methods execute synchronously while non-blocking methods execute asynchronously.

```
// Blocking
const fs = require('fs');
const data = fs.readFileSync('/file.md'); // blocks here until file is read
console.log(data);
moreWork(); // will run after console.log

// Non-blocking
const fs = require('fs');
fs.readFile('/file.md', (err, data) => {
  if (err) throw err;
  console.log(data);
});

   moreWork(); // will run before console.log
```

Non Blocking Code

Create a file input.txt with the following code:

```
 Codemithra is giving self learning content on AWS, Aptitude and MERN stack
to teach the world in simple and easy way!!!!!
```

Create a file main.js with the following code:

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

fs.readFile('input.txt', function (err, data) {
   if (err) return console.error(err);
   console.log(data.toString());
});

console.log("Program Ended");
```

Now execute the program and check the output:

```
Program Ended
Codemithra is giving self learning content on AWS, Aptitude and MERN stack to teach the world in simple and easy
way!!!!!
```

Buffers

- Node provides Buffer class which provides instances to store raw data similar to an array of integers but corresponds to a raw memory allocation outside the V8 heap.
- Buffer class is a global class that can be accessed in an application without importing the buffer module.

Writing to Buffers

Node Buffer can be constructed in a variety of ways.

Method 1
Following is the syntax to create an uninitiated Buffer of 10 octets –

```
var buf = new Buffer(10);
```

Method 2
Following is the syntax to create a Buffer from a given array –

```
var buf = new Buffer([10, 20, 30, 40, 50]);
```

Method 3
Following is the syntax to create a Buffer from a given string and optionally encoding type –

```
var buf = new Buffer("Simply Easy Learning", "utf-8");
```

Though "utf8" is the default encoding, you can use any of the following encodings "ascii", "utf8", "utf16le", "ucs2", "base64" or "hex".

Creating Buffers

Syntax

- Following is the syntax of the method to write into a Node Buffer –
- `buf.write(string[, offset][, length][, encoding])`

Parameters

- Here is the description of the parameters used –
- string – This is the string data to be written to buffer.
- offset – This is the index of the buffer to start writing at. Default value is 0.
- length – This is the number of bytes to write. Defaults to buffer.length.
- encoding – Encoding to use. 'utf8' is the default encoding.

Return Value

- This method returns the number of octets written. If there is not enough space in the buffer to fit the entire string, it will write a part of the string.

Example

```
buf = new Buffer(256);
len = buf.write("Simply Easy Learning");
console.log("Octets written : "+  len);
```

Reading from Buffers

- **Syntax**
- Following is the syntax of the method to read data from a Node Buffer –
- `buf.toString([encoding][, start][, end])`

Parameters

Here is the description of the parameters used –

- encoding – Encoding to use. 'utf8' is the default encoding.
- start – Beginning index to start reading, defaults to 0.
- end – End index to end reading, defaults is complete buffer.

Return Value

- This method decodes and returns a string from buffer data encoded using the specified character set encoding.

Example

```
buf = new Buffer(26);
for (var i = 0 ; i < 26 ; i++) {
  buf[i] = i + 97;
}

console.log( buf.toString('ascii'));       // outputs: abcdefghijklmnopqrstuvwxyz
console.log( buf.toString('ascii',0,5));   // outputs: abcde
console.log( buf.toString('utf8',0,5));    // outputs: abcde
console.log( buf.toString(undefined,0,5)); // encoding defaults to 'utf8', outputs abcde
```

Convert Buffer to JSON

Syntax
Following is the syntax of the method to convert a Node Buffer into JSON object –

```
buf.toJSON()
```

Return Value: This method returns a JSON-representation of the Buffer instance.

Example:

```
var buf = new Buffer('Simply Easy Learning');
var json = buf.toJSON(buf);
console.log(json);
```

Concatenate Buffers

Syntax:
Following is the syntax of the method to concatenate Node buffers to a single Node Buffer −

```
Buffer.concat(list[, totalLength])
```

Parameters
Here is the description of the parameters used −
list − Array List of Buffer objects to be concatenated.
totalLength − This is the total length of the buffers when concatenated.

Return Value: This method returns a Buffer instance.

Example:

```
var buffer1 = new Buffer('Ethnus');
var buffer2 = new Buffer('Codemithra');
var buffer3 = Buffer.concat([buffer1,buffer2]);
console.log("buffer3 content: " + buffer3.toString());
```

Compare Buffers

Syntax:

- Following is the syntax of the method to compare two Node buffers –
- `buf.compare(otherBuffer);`

Parameters:

- Here is the description of the parameters used –
- otherBuffer – This is the other buffer which will be compared with buf
- **Return Value:**Returns a number indicating whether it comes before or after or is the same as the otherBuffer in sort order.

Example

```
var buffer1 = new Buffer('ABC');
var buffer2 = new Buffer('ABCD');
var result = buffer1.compare(buffer2);
if(result < 0) {
   console.log(buffer1 +" comes before " + buffer2);
} else if(result === 0) {
   console.log(buffer1 +" is same as " + buffer2);
} else {
   console.log(buffer1 +" comes after " + buffer2);
}
```

Copy Buffer

Syntax:

Following is the syntax of the method to copy a node buffer –

```
buf.copy(targetBuffer[, targetStart][, sourceStart][, sourceEnd])
```

Parameters

Here is the description of the parameters used–

- targetBuffer – Buffer object where buffer will be copied.
- targetStart – Number, Optional, Default: 0
- sourceStart – Number, Optional, Default: 0
- sourceEnd – Number, Optional, Default: buffer.length

Return Value

No return value. Copies data from a region of this buffer to a region in the target buffer even if the target memory region overlaps with the source.

If undefined, the targetStart and sourceStart parameters default to 0, while sourceEnd defaults to buffer.length.

Example

```
var buffer1 = new Buffer('ABC');

//copy a buffer
var buffer2 = new Buffer(3);
buffer1.copy(buffer2);

console.log("buffer2 content: " + buffer2.toString());
```

Slice Buffer

Syntax

Following is the syntax of the method to get a sub-buffer of a node buffer –

```
buf.slice([start][, end])
```

Parameters

Here is the description of the parameters used –

- start – Number, Optional, Default: 0
- end – Number, Optional, Default: buffer.length

Return Value

Returns a new buffer which references the same memory as the old one, but offset and cropped by the start (defaults to 0) and end (defaults to buffer.length) indexes. Negative indexes start from the end of the buffer.

Example

```
var buffer1 = new Buffer('EthnusCodemithra');

//slicing a buffer
var buffer2 = buffer1.slice(0,9);
console.log("buffer2 content: " + buffer2.toString());
```

Buffer Length

Syntax

Following is the syntax of the method to get a size of a node buffer in bytes −

```
buf.length;
```

Return Value: Returns the size of a buffer in bytes.

Example

```
var buffer = new Buffer('EthnusCodemithra');
//length of the buffer
console.log("buffer length: " + buffer.length);
```

Utility Modules

S.No	Module and Description
1	**OS Module:**Provides basic operating-system related utility functions.
2	**Path Module:**Provides utilities for handling and transforming file paths.
3	**Net Module:**Provides both servers and clients as streams. Acts as a network wrapper.
4	**DNS Module:**Provides functions to do actual DNS lookup as well as to use underlying operating system name resolution functionalities.
5	**Domain Module:**Provides ways to handle multiple different I/O operations as a single group.

What is NPM?

NPM stands for Node Package Manager.

Node Package Manager (NPM) provides two main functionalities –

- Online repositories for node.js packages/modules which are searchable on search.nodejs.org
- Command line utility to install Node.js packages, do version management and dependency management of Node.js packages.

To verify that npm is being installed in your system just type:

```
C:\Users\welcome\Documents>npm -v
```

The output will appear as 6.14.4 whereas 6.14.4 is the current version of Node.js

What is Package?

- A package in Node.js contains all the files you need for a module.
- Modules are JavaScript libraries you can include in your project.

Installing modules in NPM

There is a simple syntax to install any Node.js module locally –

```
C:\Users\welcome\Documents>npm install <Module Name>
```

For example, following is the command to install a famous Node.js web framework module called express –

```
C:\Users\welcome\Documents>npm install express
```

Now you can use this module in your js file as following –

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

Adding dependencies in package.json file

package.json is present in the root directory of any Node application/module and is used to define the properties of a package.

Let's open package.json of express package present in `node_modules/express/`.

```
{      "name": "express",
       "description": "Fast, unopinionated, minimalist web framework",
       "version": "4.11.2",
       "author": {  "name": "TJ Holowaychuk",
               "email": "tj@vision-media.ca"   },
       "contributors": [{
       "name": "Aaron Heckmann",
       "email": "aaron.heckmann+github@gmail.com"
   },
```

Adding dependencies

```
{       "name": "Ciaran Jessup",
        "email": "ciaranj@gmail.com"    },
{       "name": "Douglas Christopher Wilson",
        "email": "doug@somethingdoug.com"    },
{       "name": "Guillermo Rauch",
        "email": "rauchg@gmail.com"    },
{       "name": "Jonathan Ong",
        "email": "me@jongleberry.com"    },
{       "name": "Roman Shtylman",
        "email": "shtylman+expressjs@gmail.com"    },
{       "name": "Young Jae Sim",
        "email": "hanul@hanul.me"   } ],
```

Adding dependencies

```
"license": "MIT", "repository": {
      "type": "git",
      "url":
"https://github.com/strongloop/express"   },
      "homepage":     "https://expressjs.com/",
"keywords": [
      "express",
      "framework",
      "sinatra",
      "web",
      "rest",
      "restful",
      "router",
      "app",
      "api"  ],
```

Adding dependencies

```
"dependencies": {
      "accepts": "~1.2.3",
      "content-disposition": "0.5.0",
      "cookie-signature": "1.0.5",
      "debug": "~2.1.1",
      "depd": "~1.0.0",
      "escape-html": "1.0.1",
      "etag": "~1.5.1",
      "finalhandler": "0.3.3",
      "fresh": "0.2.4",
      "media-typer": "0.3.0",
      "methods": "~1.1.1",
      "on-finished": "~2.2.0",
      "parseurl": "~1.3.0",
      "path-to-regexp": "0.1.3",
```

Adding dependencies

```
"proxy-addr": "~1.0.6",
      "qs": "2.3.3",
      "range-parser": "~1.0.2",
      "send": "0.11.1",
      "serve-static": "~1.8.1",
      "type-is": "~1.5.6",
      "vary": "~1.0.0",
      "cookie": "0.1.2",
      "merge-descriptors": "0.0.2",
      "utils-merge": "1.0.0"
   },
```

Uninstalling a module

Use the following command to uninstall a Node.js module.

```
$ npm uninstall express
```

Once NPM uninstalls the package, you can verify it by looking at the content of /node_modules/ directory or type the following command –

```
$ npm ls
```

Event Emitter Class

Many objects in a Node emit events, for example, a net.Server emits an event each time a peer connects to it, an fs.readStream emits an event when the file is opened. All objects which emit events are the instances of events.EventEmitter.EventEmitter class lies in the events module. It is accessible via the following code –

```
// Import events module
var events = require('events');

// Create an eventEmitter object
var eventEmitter = new events.EventEmitter();
```

When an EventEmitter instance faces any error, it emits an 'error' event. When a new listener is added, 'newListener' event is fired and when a listener is removed, 'removeListener' event is fired.

EventEmitter provides multiple properties like on and emit. on property is used to bind a function with the event and emit is used to fire an event.

Methods

1.addListener(event, listener): Adds a listener at the end of the listeners array for the specified event. No checks are made to see if the listener has already been added. Multiple calls passing the same combination of event and listener will result in the listener being added multiple times. Returns emitter, so calls can be chained.

2.on(event, listener): Adds a listener at the end of the listeners array for the specified event. No checks are made to see if the listener has already been added. Multiple calls passing the same combination of event and listener will result in the listener being added multiple times. Returns emitter, so calls can be chained.

3.once(event, listener): Adds a one time listener to the event. This listener is invoked only the next time the event is fired, after which it is removed. Returns emitter, so calls can be chained.

4.removeListener(event, listener): Removes a listener from the listener array for the specified event. Caution − It changes the array indices in the listener array behind the listener. removeListener will remove, at most, one instance of a listener from the listener array. If any single listener has been added multiple times to the listener array for the specified event, then removeListener must be called multiple times to remove each instance. Returns emitter, so calls can be chained.

Methods

5.removeAllListeners([event]): Removes all listeners, or those of the specified event. It's not a good idea to remove listeners that were added elsewhere in the code, especially when it's on an emitter that you didn't create (e.g. sockets or file streams). Returns emitter, so calls can be chained.

6.setMaxListeners(n): By default, EventEmitters will print a warning if more than 10 listeners are added for a particular event. This is a useful default which helps finding memory leaks. Obviously not all Emitters should be limited to 10. This function allows that to be increased. Set to zero for unlimited.

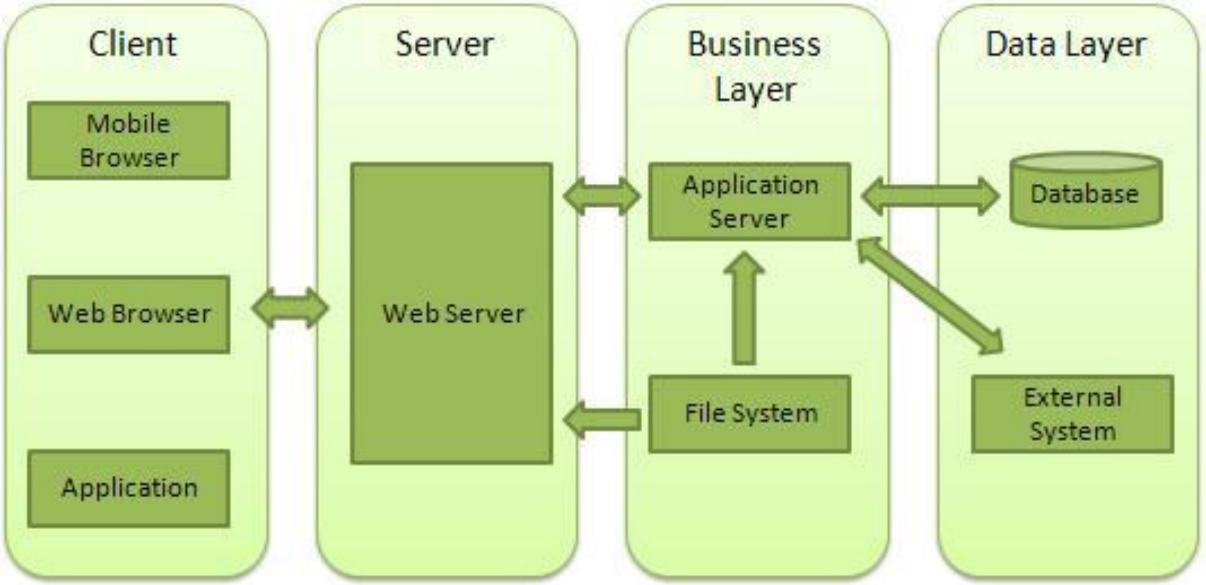
7.listeners(event): Returns an array of listeners for the specified event.

8.emit(event, [arg1], [arg2], [...]): Execute each of the listeners in order with the supplied arguments. Returns true if the event had listeners, false otherwise.

What is a Web Server?

- A Web Server is a software application which handles HTTP requests sent by the HTTP client, like web browsers, and returns web pages in response to the clients. Web servers usually deliver html documents along with images, style sheets, and scripts.
- Most of the web servers support server-side scripts, using scripting languages or redirecting the task to an application server which retrieves data from a database and performs complex logic and then sends a result to the HTTP client through the Web server.

Web application architecture



- **Client** – This layer consists of web browsers, mobile browsers or applications which can make HTTP requests to the web server.
- **Server** – This layer has the Web server which can intercept the requests made by the clients and pass them the response.
- **Business** – This layer contains the application server which is utilized by the web server to do the required processing. This layer interacts with the data layer via the database or some external programs.
- **Data** – This layer contains the databases or any other source of data.

Creating a web server using node js

Node.js provides an **http** module which can be used to create an HTTP client of a server. Following is the bare minimum structure of the HTTP server which listens at 8081 port.

File: server.js

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

// Create a server
http.createServer( function (request, response) {
   // Parse the request containing file name
   var pathname = url.parse(request.url).pathname;

   // Print the name of the file for which request is made.
   console.log("Request for " + pathname + " received.");

   // Read the requested file content from file system
   fs.readFile(pathname.substr(1), function (err, data) {
      if (err) {
         console.log(err);

         // HTTP Status: 404 : NOT FOUND
         // Content Type: text/plain
         response.writeHead(404, {'Content-Type':
'text/html'});
      }
else {
         //Page found
         // HTTP Status: 200 : OK
         // Content Type: text/plain
         response.writeHead(200, {'Content-Type':
'text/html'});

         // Write the content of the file to response body
         response.write(data.toString());
      }

      // Send the response body
      response.end();
   });
}).listen(8081);

// Console will print the message
console.log('Server running at http://127.0.0.1:8081/');
```

File: index.html

```
<html>
   <head>
      <title>Sample Page</title>
   </head>

   <body>
      Hello World!
   </body>
</html>
```

Now let us run the server.js to see the result −

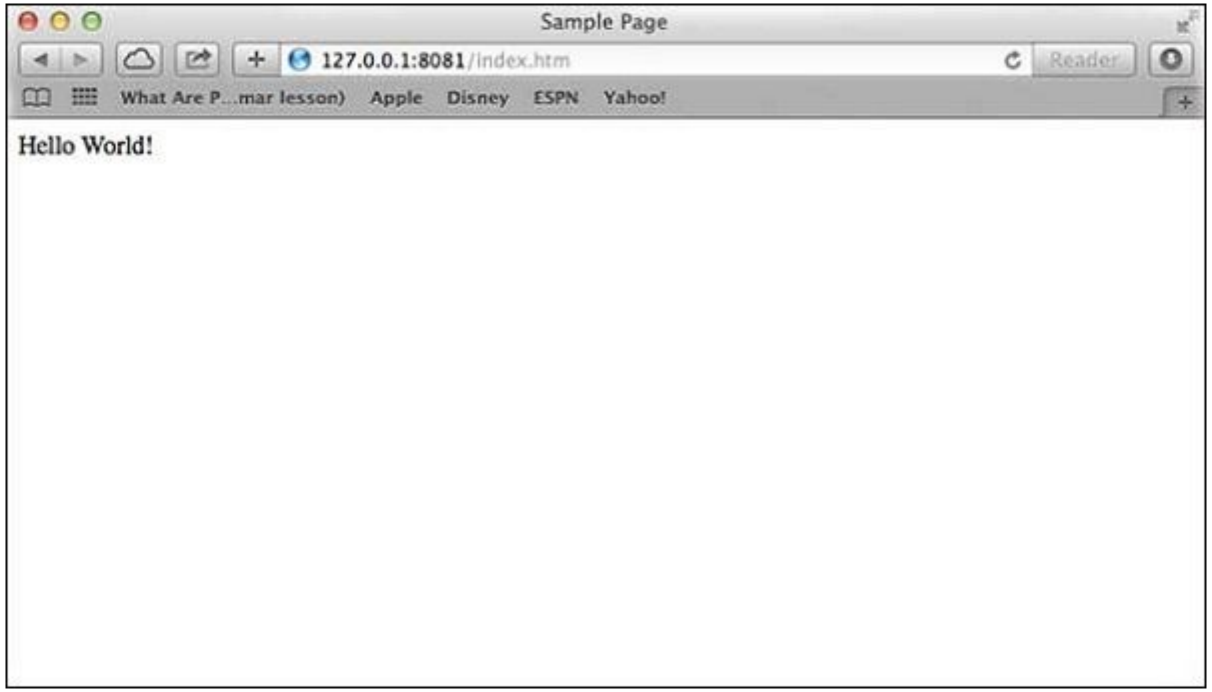
```
$ node server.js
```

Verify the Output.

```
Server running at http://127.0.0.1:8081/
```

Make a request to Node.js server

Make a request to Node.js server
Open http://127.0.0.1:8081/index.htm in any browser to see the following result.



Hello World!

First Server Application
Verify the Output at server end.

```
Server running at http://127.0.0.1:8081/
Request for /index.htm received.
```

Make a request to Node.js server

File: client.js

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

// Options to be used by request
var options = {
   host: 'localhost',
   port: '8081',
   path: '/index.htm'
};

// Callback function is used to deal with response
var callback = function(response) {
   // Continuously update stream with data
   var body = '';
   response.on('data', function(data) {
      body += data;
   });
   response.on('end', function() {
      // Data received completely.
      console.log(body);
   });
}
// Make a request to the server
var req = http.request(options, callback);
req.end();
```

Now run the client.js from a different command terminal other than server.js to see the result –

```
$ node client.js
```

Output

Verify the Output.

```
<html>
   <head>
      <title>Sample Page</title>
   </head>

   <body>
      Hello World!
   </body>
</html>
```

Verify the Output at server end.

```
Server running at http://127.0.0.1:8081/
Request for /index.htm received.
```