

THE NEW COLLEGE (AUTONOMOUS), CHENNAI-14.
DEPARTMENT OF COMPUTER APPLICATIONS – SHIFT - II
STUDY MATERIAL

SUBJECT: OPEN SOURCE TECHNOLOGIES

SUB CODE: 20BHM615

CLASS: III BCA

STAFF: Dr. K. SANKAR

UNIT-II

1. NODE.JS

- 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. Node.js 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 is an open source, cross-platform runtime environment for developing server-side and networking applications. Node.js applications are written in JavaScript, and can be run within the Node.js runtime on OS X, Microsoft Windows, and Linux.
- 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

2. WORKING OF NODE.JS

Node.js accepts the request from the clients and sends the response, while working with the request node.js handles them with a single thread. To operate I/O operations or requests node.js use the concept of threads. Thread is a sequence of instructions that the server needs to perform. It runs parallel on the server to provide the information to multiple clients. Node.js is an event loop single-threaded language. It can handle concurrent requests with a single thread without blocking it for one request.

Node.js basically works on two concepts

- Non-blocking I/O
- Asynchronous

Non-blocking I/O: Non-blocking I/O means working with multiple requests without blocking the thread for a single request. I/O basically interacts with external systems such as files, databases. Node.js is not used for CPU-intensive work means for calculations, video processing because a single thread cannot handle the CPU works.

Asynchronous: Asynchronous is executing a call back function. The moment we get the response from the other server or database it will execute a call back function. Call-back functions are called as soon as some work is finished and this is because the node.js uses an event-driven architecture. The single thread doesn't work with the request instead it sends the request to another system which resolves the request and it is accessible for another request.

It has a strong focus on asynchronous and I/O, this gives node access to the underlying computer operating system, file system, and networking. Libuv implements two extremely important **features of node.js**

- Event loop
- Thread pool

Event loop: The event loop contains a single thread and is responsible for handling easy tasks like executing call-backs and network I/O. When the program is to initialize all the top-level code is executed, the code is not in the call back function. All the applications code that is inside call back functions will run in the event loop. Event Loop is the heart of node.js. When we start our node application the event loop starts running right away. Most of the work is done in the event loop.

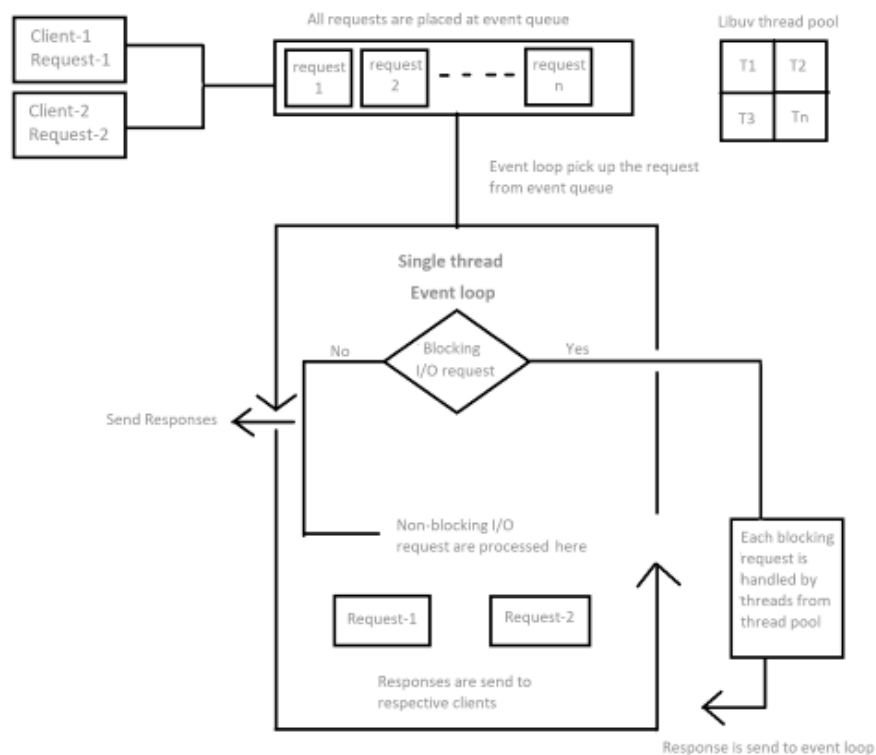
Nodejs use event-driven-architecture.

- Events are emitted.
- Event loop picks them up.
- Call-backs are called.

Event queue: As soon as the request is sent the thread places the request into a queue. It is known as an event queue. The process like app receiving HTTP request or server or a timer will emit event as soon as they are done with the work and event loop will pick up these events and call the call back functions that are associated with each event and response is sent to the

client. The event loop is an indefinite loop that continuously receives the request and processes them. It checks the queue and waits for the incoming request indefinitely.

Thread pool: Though node.js is single-threaded it internally maintains a thread pool. When non-blocking requests are accepted there are processed in an event loop, but while accepting blocking requests it checks for available threads in a thread pool, assigns a thread to the client's request which is then processed and send back to the event loop, and response is sent to the respective client.



3. FEATURES OF NODE.JS

Following are some of the important features that make Node.js the first choice of software architects.

- **Asynchronous and Event Driven** – All APIs of Node.js library are asynchronous, that is, non-blocking. It essentially means a Node.js based server never waits for an API to return data. The server moves to the next API after calling it and a notification mechanism of Events of Node.js helps the server to get a response from the previous API call.
- **Very Fast** – Being built on Google Chrome's V8 JavaScript Engine, Node.js library is very fast in code execution.

- **Single Threaded but Highly Scalable** – Node.js uses a single threaded model with event looping. Event mechanism helps the server to respond in a non-blocking way and makes the server highly scalable as opposed to traditional servers which create limited threads to handle requests. Node.js uses a single threaded program and the same program can provide service to a much larger number of requests than traditional servers like Apache HTTP Server.
- **No Buffering** – Node.js applications never buffer any data. These applications simply output the data in chunks.
- **License** – Node.js is released under the MIT license

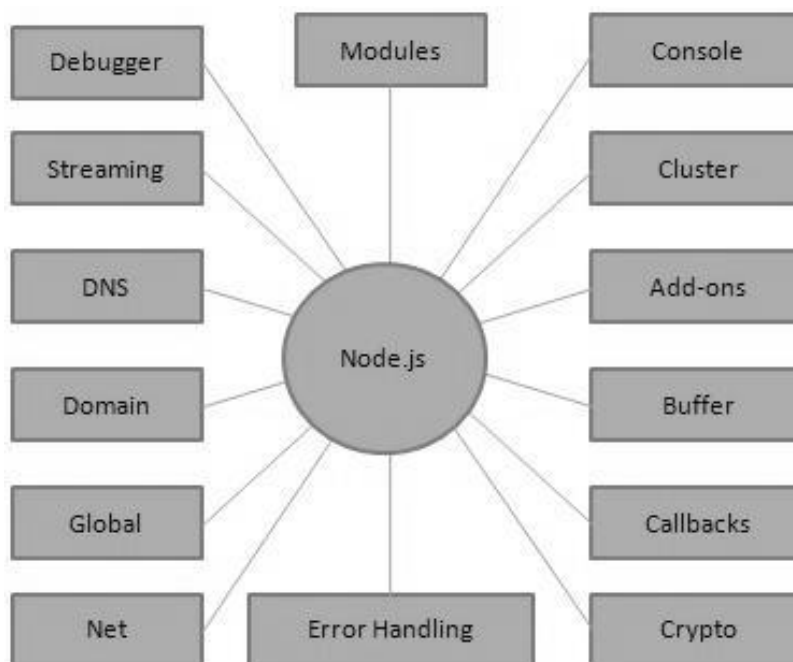
4. WHO USES NODE.JS?

Following is the link on github wiki containing an exhaustive list of projects, application and companies which are using Node.js. This list includes eBay, General Electric, GoDaddy, Microsoft, PayPal, Uber, Wikipins, Yahoo!, and Yammer to name a few.

- Projects, Applications, and Companies Using Node

Concepts

The following diagram depicts some important parts of Node.js which we will discuss in detail in the subsequent chapters.



Where to Use Node.js?

Following are the areas where Node.js is proving itself as a perfect technology partner.

- I/O bound Applications
- Data Streaming Applications
- Data Intensive Real-time Applications (DIRT)
- JSON APIs based Applications
- Single Page Applications

5. NODE PROGRAMMING

- Node.js is an open source server environment
- Node.js is free
- Node.js runs on various platforms (Windows, Linux, Unix, Mac OS X, etc.)
- Node.js uses JavaScript on the server

Example

```
var http = require('http');
http.createServer(function (req, res) {
  res.writeHead(200, {'Content-Type': 'text/plain'});
  res.end('Hello World!');
}).listen(8080);
```

Output

Hello World!

6. 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, 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.

Online REPL Terminal

To simplify your learning, we have set up an easy to use Node.js REPL environment online, where you can practice Node.js syntax – [Launch Node.js REPL Terminal](#)

Starting REPL

REPL can be started by simply running **node** on shell/console without any arguments as follows.

```
$ node
```

You will see the REPL Command prompt **>** where you can type any Node.js command –

```
$ node
```

```
>
```

Simple Expression

Let's try a simple mathematics at the Node.js REPL command prompt –

```
$ node
```

```
> 1 + ( 2 * 3 ) - 4
```

```
3
```

```
>
```

Use Variables

You can make use variables to store values and print later like any conventional script. If **var** keyword is not used, then the value is stored in the variable and printed. Whereas if **var** keyword is used, then the value is stored but not printed. You can print variables using **console.log()**.

```
$ node
```

```
> x = 10
```

```
10
```

```
> var y = 10
```

```
Undefined
```

```
> x + y
```

```
20
```

```
> console.log ("Hello World")
```

```
Hello World
```

```
Undefined
```

Multiline Expression

Node REPL supports multiline expression similar to JavaScript. Let's check the following do-while loop in action –

```
$ node
> var x = 0
undefined
> do {
  ... x++;
  ... console.log("x: " + x);
  ... }
while ( x < 5 );
x: 1
x: 2
x: 3
x: 4
x: 5
undefined
>
```

... comes automatically when you press Enter after the opening bracket. Node automatically checks the continuity of expressions.

Underscore Variable

You can use underscore (_) to get the last result –

```
$ node
> var x = 10
undefined
> var y = 20
undefined
> x + y
30
> var sum = _
undefined
```

```
> console.log(sum)
```

```
30
```

```
undefined
```

```
>
```

7. NODE JS ENVIRONMENT SETUP - NODE.JS INSTALLATION

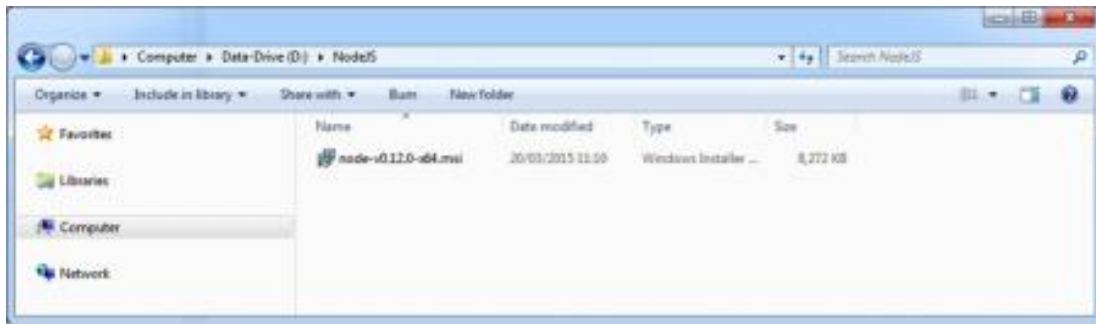
1. Access Node JS Official Website “<https://nodejs.org/en/>”



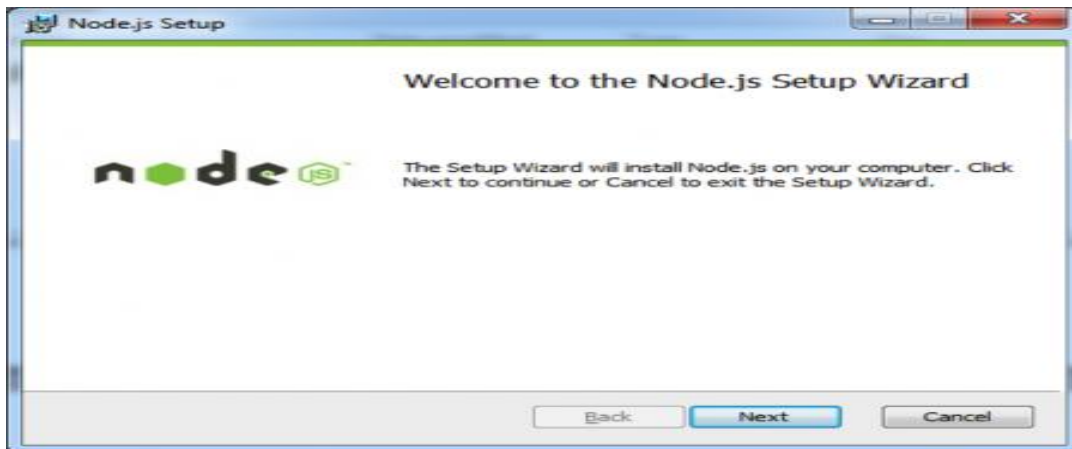
2. Here we can find “INSTALL” button. If we want to download latest stable version, we can click on this button. If we want to select Node JS Platform based on our Hardware and Software requirements like Windows 32bit or 64bit OS, please click on “Downloads” button and select your required Node JS version to download.



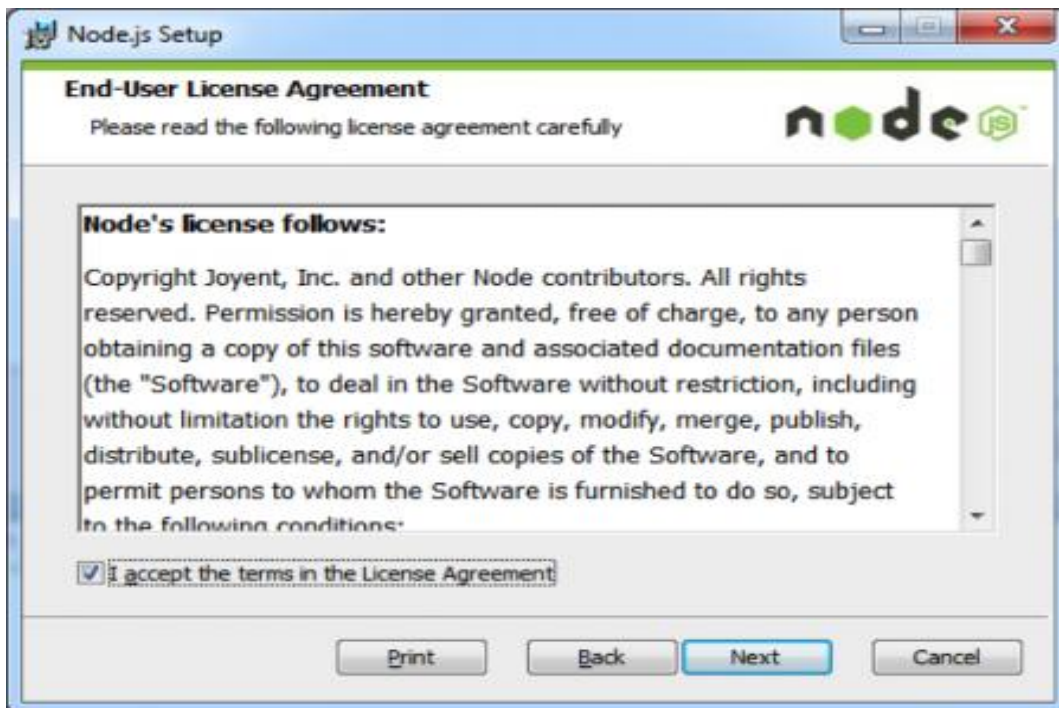
3. Please click on “INSTALL” button to download latest Node JS Platform Version 0.12.0, this is the latest version as of writing this post.



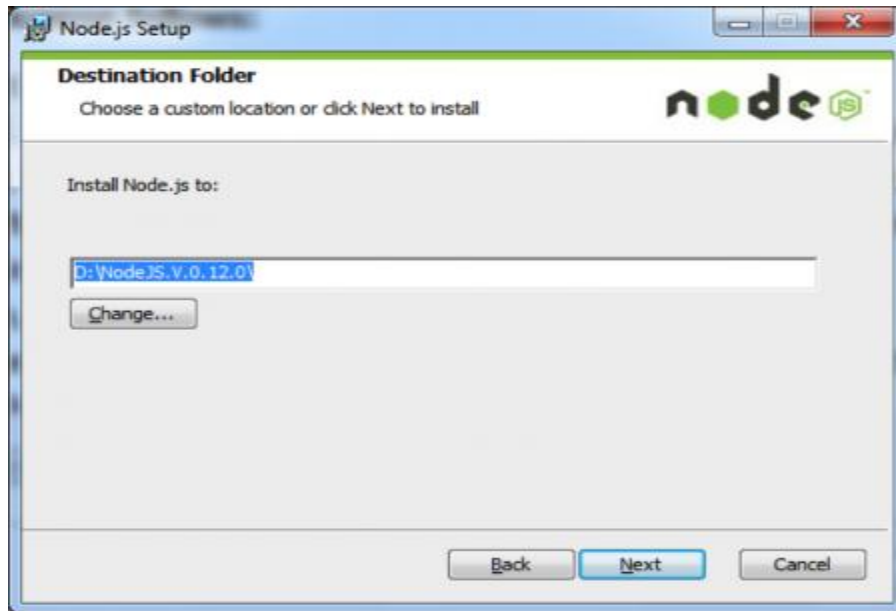
4. Please double click on “node-v0.12.0-x64.msi” file to start Installation process.



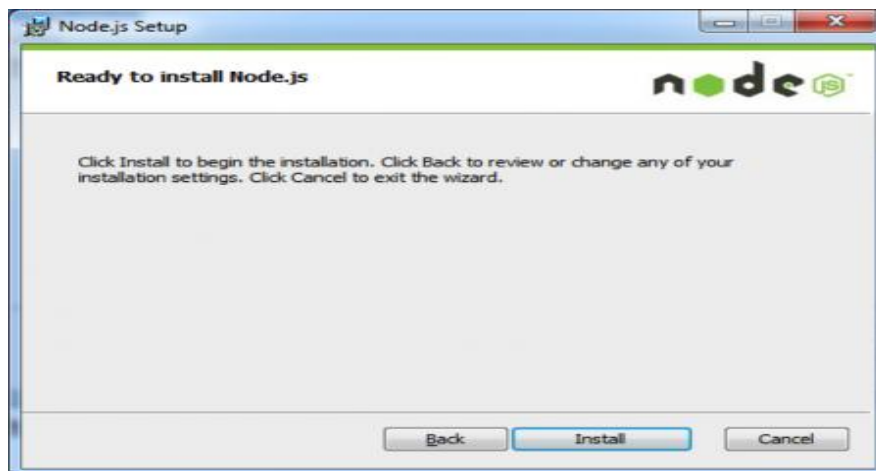
5. Click on “Next” Button. Accept License Agreement and Click on “Next” Button.



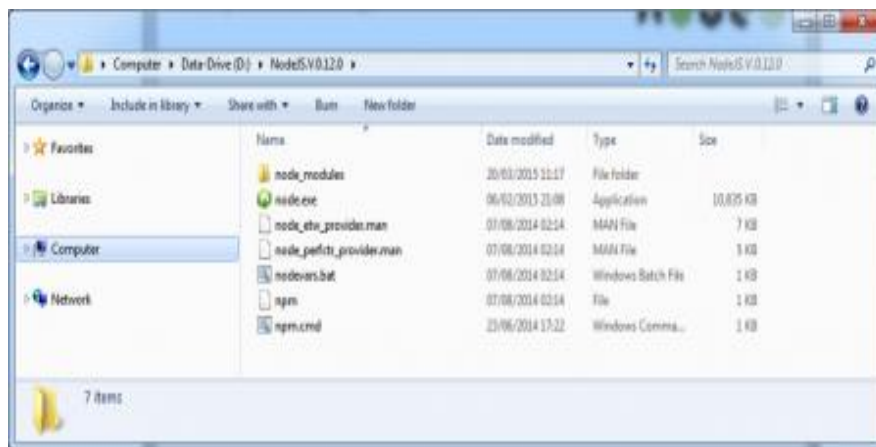
6. Choose your required location to install.



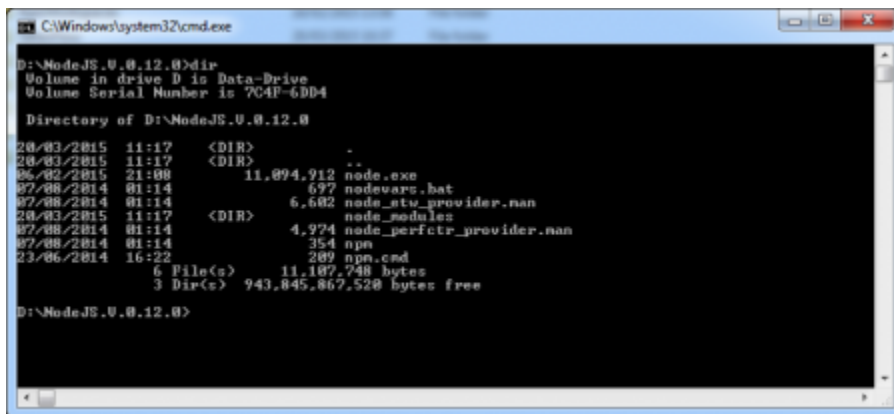
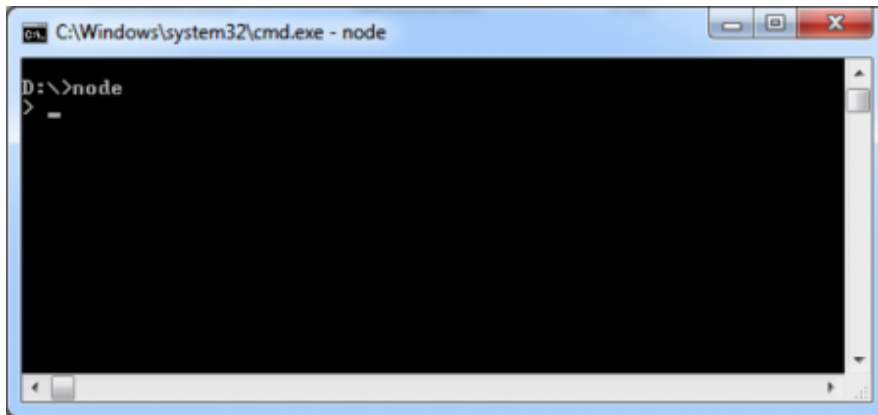
7. Choose Default values and click on “Install” Button.



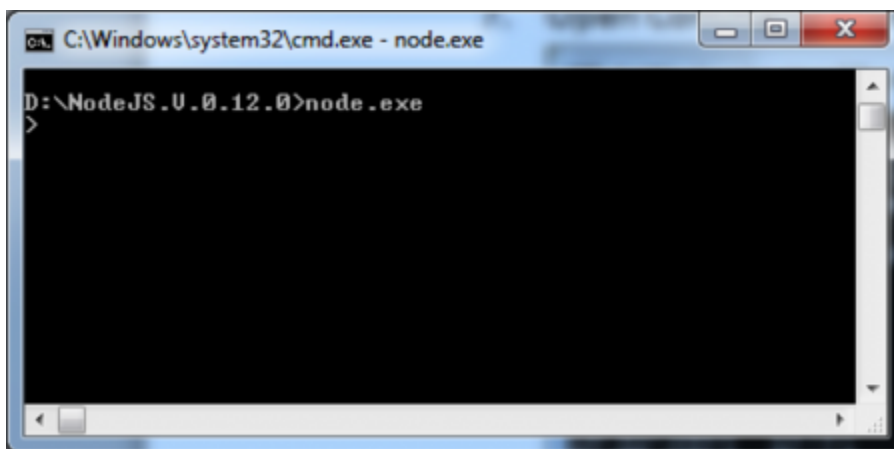
8. Open Node JS Platform Home in your Explorer.



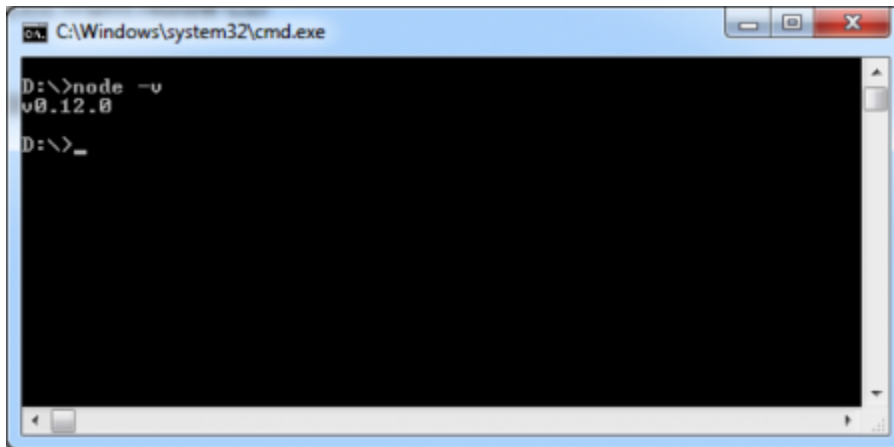
9. Open Command prompt at NODSJS_HOME or anywhere like D:\



10. Run “node.exe” comma

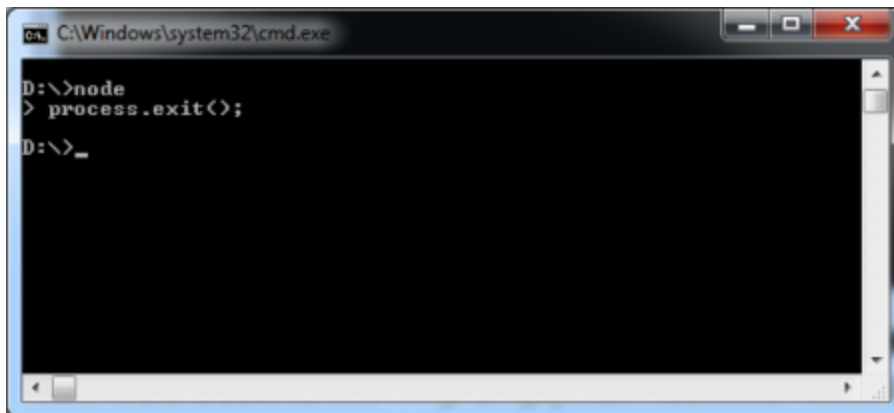


11. Now we are able to see “>” prompt, that means our Node JS Base Environment Setup is done.
12. Check Node JS Version: Use the following command to know your Node JS Version from Command prompt. node -v



```
C:\Windows\system32\cmd.exe
D:\>node -v
v0.12.0
D:\>_
```

13. Exit from Node CLI: We can use “process.exit()” command to exit from Node CLI.



```
C:\Windows\system32\cmd.exe
D:\>node
> process.exit();
D:\>_
```

14. We can also use **Ctrl + D** OR use **Ctrl + C Twice** to exit from Node CLI.

8. NODE PACKAGE MANAGER (NPM)

It 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.

NPM comes bundled with Node.js installable after v0.6.3 version. To verify the same, open console and type the following command and see the result –

```
$ npm -version 2.7.1
```

Installing Modules using NPM

Syntax to install any Node.js module –

```
$ npm install <Module Name>
```

Example,

\$ npm install express

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

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

9. CALLBACKS CONCEPT

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.

Blocking Code Example

```
var fs = require("fs");
var data = fs.readFileSync('input.txt');
console.log(data.toString());
console.log("Program Ended");
```

Verify the Output.

Program Ended

Non-Blocking Code Example

```
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 run the main.js to see the result –

\$ node main.js

Verify the Output.

Program Ended

10. EVENT LOOP

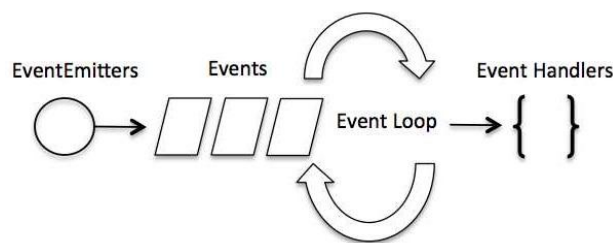
Node.js is a single-threaded application, but it can support concurrency via the concept of **event** and **callbacks**. Every API of Node.js is asynchronous and being single-threaded, they

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**.

Example

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

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

// Create an event handler as follows
var connectHandler = function connected() {
  console.log('connection succesful.');
```

// Fire the data_received event

```
  eventEmitter.emit('data_received');
}
```

// 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 succesfully.');
```

```
// Fire the connection event
```

```
eventEmitter.emit('connection');  
console.log("Program Ended.");
```

Output

```
$ node main.js
```

```
connection successful.
```

```
data received successfully.
```

```
Program Ended.
```

11. EVENT EMITTER

Many objects in a Node emit events, for example, a `net.Server` emits an event each time a peer connects to it, and an `fs.readStream` emits an event when the file is opened.

EventEmitter Class

As we have seen in the previous section, `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();
```

12. 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.

Creating 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");
```

Writing to Buffers

Syntax

```
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);
```

Output

```
Octets written : 20
```

Reading from Buffers

Syntax

```
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
```

Output

```
abcdefghijklmnopqrstuvwxyz
abcde
abcde
abcde
```

13. STREAMS

Streams are objects that let you read data from a source or write data to a destination in continuous fashion.

Types of streams

In Node.js, there are four types of streams –

- **Readable** – Stream which is used for read operation.
- **Writable** – Stream which is used for write operation.
- **Duplex** – Stream which can be used for both read and write operation.
- **Transform** – A type of **duplex stream** where the output is computed based on input.

Reading from a Stream

Create a js file named main.js with the following code –

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

// Create a readable stream
var readerStream = fs.createReadStream('input.txt');
// Set the encoding to be utf8.
readerStream.setEncoding('UTF8');
// Handle stream events --> data, end, and error
readerStream.on('data', function(chunk) {
    data += chunk;
});
readerStream.on('end',function() {
    console.log(data);
});
readerStream.on('error', function(err) {
    console.log(err.stack);
});
console.log("Program Ended");
```

Output

Program Ended

Writing to a Stream

```
var fs = require("fs");
var data = 'Simply Easy Learning';
// Create a writable stream
var writerStream = fs.createWriteStream('output.txt');
// Write the data to stream with encoding to be utf8
writerStream.write(data,'UTF8');
// Mark the end of file
writerStream.end();
// Handle stream events --> finish, and error
writerStream.on('finish', function() {
```

```
    console.log("Write completed.");  
  });  
  writerStream.on('error', function(err) {  
    console.log(err.stack);  
  });  
  console.log("Program Ended");
```

Output

Program Ended
Write completed.

Piping the Streams

Piping is a mechanism where we provide the output of one stream as the input to another stream. It is normally used to get data from one stream and to pass the output of that stream to another stream. There is no limit on piping operations. Now we'll show a piping example for reading from one file and writing it to another file.

```
var fs = require("fs");  
// Create a readable stream  
var readerStream = fs.createReadStream('input.txt');  
// Create a writable stream  
var writerStream = fs.createWriteStream('output.txt');  
// Pipe the read and write operations  
// read input.txt and write data to output.txt  
readerStream.pipe(writerStream);  
console.log("Program Ended");
```

Output

Program Ended

Chaining the Streams

Chaining is a mechanism to connect the output of one stream to another stream and create a chain of multiple stream operations. It is normally used with piping operations. Now we'll use piping and chaining to first compress a file and then decompress the same.

```
var fs = require("fs");
var zlib = require('zlib');
// Compress the file input.txt to input.txt.gz
fs.createReadStream('input.txt')
  .pipe(zlib.createGzip())
  .pipe(fs.createWriteStream('input.txt.gz'));
console.log("File Compressed.");
```

Output

File Compressed.

14. FILE SYSTEM

Node implements File I/O using simple wrappers around standard POSIX functions. The Node

Syntax

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

Synchronous vs Asynchronous

Asynchronous methods take the last parameter as the completion function callback and the first parameter of the callback function as error. It is better to use an asynchronous method instead of a synchronous method, as the former never blocks a program during its execution, whereas the second one does.

Example

```
var fs = require("fs");
// Asynchronous read
fs.readFile('input.txt', function (err, data) {
  if (err) {
    return console.error(err);
  }
  console.log("Asynchronous read: " + data.toString());
});
// Synchronous read
var data = fs.readFileSync('input.txt');
console.log("Synchronous read: " + data.toString());
console.log("Program Ended");
```

Output

Program Ended

Open a File

Syntax

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

Parameters

Here is the description of the parameters used –

- **path** – This is the string having file name including path.
- **flags** – Flags indicate the behavior of the file to be opened. All possible values have been mentioned below.
- **mode** – It sets the file mode (permission and sticky bits), but only if the file was created. It defaults to 0666, readable and writeable.
- **callback** – This is the callback function which gets two arguments (err, fd).

Flags

Flags for read/write operations are –

Sr.No.	Flag & Description
1	r Open file for reading. An exception occurs if the file does not exist.
2	r+ Open file for reading and writing. An exception occurs if the file does not exist.
3	rs Open file for reading in synchronous mode.
4	rs+ Open file for reading and writing, asking the OS to open it synchronously. See notes for 'rs' about using this with caution.
5	W Open file for writing. The file is created (if it does not exist) or truncated (if it exists).
6	Wx Like 'w' but fails if the path exists.
7	w+ Open file for reading and writing. The file is created (if it does not exist) or truncated (if it exists).
8	wx+ Like 'w+' but fails if path exists.

9	a Open file for appending. The file is created if it does not exist.
10	ax Like 'a' but fails if the path exists.
11	a+ Open file for reading and appending. The file is created if it does not exist.
12	ax+ Like 'a+' but fails if the the path exists.

Example

```
var fs = require("fs");
// Asynchronous - Opening File
console.log("Going to open file!");
fs.open('input.txt', 'r+', function(err, fd) {
  if (err) {
    return console.error(err);
  }
  console.log("File opened successfully!");
});
```

Now run the main.js to see the result –

```
$ node main.js
```

Verify the Output.

Going to open file!

File opened successfully!

Get File Information

Syntax

```
fs.stat(path, callback)
```

Parameters

- **path** – This is the string having file name including path.
- **callback** – This is the callback function which gets two arguments (err, stats) where **stats** is an object of fs.Stats type which is printed below in the example.

These methods are given in the following table.

Sr.No.	Method & Description
--------	----------------------

1	stats.isFile() Returns true if file type of a simple file.
2	stats.isDirectory() Returns true if file type of a directory.
3	stats.isBlockDevice() Returns true if file type of a block device.
4	stats.isCharacterDevice() Returns true if file type of a character device.
5	stats.isSymbolicLink() Returns true if file type of a symbolic link.
6	stats.isFIFO() Returns true if file type of a FIFO.
7	stats.isSocket() Returns true if file type of a socket.

Example

```
var fs = require("fs");
console.log("Going to get file info!");
fs.stat('input.txt', function (err, stats) {
  if (err) {
    return console.error(err);
  }
  console.log(stats);
  console.log("Got file info successfully!");
  // Check file type
  console.log("isFile ? " + stats.isFile());
  console.log("isDirectory ? " + stats.isDirectory());
});
```

Now run the main.js to see the result –

```
$ node main.js
```

Verify the Output.

Going to get file info!

```
{
  dev: 1792,
  mode: 33188,
  nlink: 1,
  uid: 48,
  gid: 48,
  rdev: 0,
  blksize: 4096,
  ino: 4318127,
  size: 97,
  blocks: 8,
  atime: Sun Mar 22 2015 13:40:00 GMT-0500 (CDT),
  mtime: Sun Mar 22 2015 13:40:57 GMT-0500 (CDT),
  ctime: Sun Mar 22 2015 13:40:57 GMT-0500 (CDT)
}
```

Got file info successfully!

isFile ? true

isDirectory ? false

Writing a File

Syntax

Following is the syntax of one of the methods to write into a file –

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

This method will over-write the file if the file already exists. If you want to write into an existing file then you should use another method available.

Parameters

Here is the description of the parameters used –

- **path** – This is the string having the file name including path.
- **data** – This is the String or Buffer to be written into the file.
- **options** – The third parameter is an object which will hold {encoding, mode, flag}. By default. encoding is utf8, mode is octal value 0666. and flag is 'w'

- **callback** – This is the callback function which gets a single parameter `err` that returns an error in case of any writing error.

Example

```
var fs = require("fs");
console.log("Going to write into existing file");
fs.writeFile('input.txt', 'Simply Easy Learning!', function(err) {
  if (err) {
    return console.error(err);
  }
  console.log("Data written successfully!");
  console.log("Let's read newly written data");
  fs.readFile('input.txt', function (err, data) {
    if (err) {
      return console.error(err);
    }
    console.log("Asynchronous read: " + data.toString());
  });
});
```

Now run the `main.js` to see the result –

```
$ node main.js
```

Verify the Output.

Reading a File

Syntax

```
fs.read(fd, buffer, offset, length, position, callback)
```

This method will use file descriptor to read the file. If you want to read the file directly using the file name, then you should use another method available.

Parameters

- **fd** – This is the file descriptor returned by `fs.open()`.
- **buffer** – This is the buffer that the data will be written to.
- **offset** – This is the offset in the buffer to start writing at.

- **length** – This is an integer specifying the number of bytes to read.
- **position** – This is an integer specifying where to begin reading from in the file. If position is null, data will be read from the current file position.
- **callback** – This is the callback function which gets the three arguments, (err, bytesRead, buffer).

Example

```
var fs = require("fs");
var buf = new Buffer(1024);
console.log("Going to open an existing file");
fs.open('input.txt', 'r+', function(err, fd) {
  if (err) {
    return console.error(err);
  }
  console.log("File opened successfully!");
  console.log("Going to read the file");
  fs.read(fd, buf, 0, buf.length, 0, function(err, bytes){
    if (err){
      console.log(err);
    }
    console.log(bytes + " bytes read");
    // Print only read bytes to avoid junk.
    if(bytes > 0){
      console.log(buf.slice(0, bytes).toString());
    }
  });
});
```

Verify the Output.

Going to read the file

97 bytes read

Closing a File

Syntax

```
fs.close(fd, callback)
```

Parameters

Here is the description of the parameters used –

- **fd** – This is the file descriptor returned by file `fs.open()` method.
- **callback** – This is the callback function. No arguments other than a possible exception are given to the completion callback.

Example

```
var fs = require("fs");
var buf = new Buffer(1024);
console.log("Going to open an existing file");
fs.open('input.txt', 'r+', function(err, fd) {
  if (err) {
    return console.error(err);
  }
  console.log("File opened successfully!");
  console.log("Going to read the file");
  fs.read(fd, buf, 0, buf.length, 0, function(err, bytes) {
    if (err) {
      console.log(err);
    }
    // Print only read bytes to avoid junk.
    if(bytes > 0) {
      console.log(buf.slice(0, bytes).toString());
    }
    // Close the opened file.
    fs.close(fd, function(err) {
      if (err) {
        console.log(err);
      }
      console.log("File closed successfully.");
    });
  });
});
```

Now runs the main.js to see the result –

```
$ node main.js
```

Output.

Going to open an existing file

File opened successfully!

Going to read the file

File closed successfully.

Truncate a File

Syntax

```
fs.ftruncate(fd, len, callback)
```

Parameters

Here is the description of the parameters used –

- **fd** – This is the file descriptor returned by fs.open().
- **len** – This is the length of the file after which the file will be truncated.
- **callback** – This is the callback function. No arguments other than a possible exception are given to the completion callback.

Example

```
var fs = require("fs");
var buf = new Buffer(1024);
console.log("Going to open an existing file");
fs.open('input.txt', 'r+', function(err, fd) {
  if (err) {
    return console.error(err);
  }
  console.log("File opened successfully!");
  console.log("Going to truncate the file after 10 bytes");
  // Truncate the opened file.
  fs.ftruncate(fd, 10, function(err) {
    if (err) {
      console.log(err);
    }
  });
});
```

```
}  
console.log("File truncated successfully.");  
console.log("Going to read the same file");  
    fs.read(fd, buf, 0, buf.length, 0, function(err, bytes){  
    if (err) {  
        console.log(err);  
    }  
    // Print only read bytes to avoid junk.  
    if(bytes > 0) {  
        console.log(buf.slice(0, bytes).toString());  
    }  
    // Close the opened file.  
    fs.close(fd, function(err) {  
        if (err) {  
            console.log(err);  
        }  
        console.log("File closed successfully.");  
    });  
});  
});  
});
```

Now run the main.js to see the result –

```
$ node main.js
```

Verify the Output.

Going to open an existing file

File opened successfully!

Going to truncate the file after 10 bytes

File truncated successfully.

Going to read the same file

Tutorials

File closed successfully.

Delete a File

Syntax

`fs.unlink(path, callback)`

Parameters

Here is the description of the parameters used –

- **path** – This is the file name including path.
- **callback** – This is the callback function. No arguments other than a possible exception are given to the completion callback.

Example

```
var fs = require("fs");
console.log("Going to delete an existing file");
fs.unlink('input.txt', function(err) {
  if (err) {
    return console.error(err);
  }
  console.log("File deleted successfully!");
});
```

Now run the `main.js` to see the result –

```
$ node main.js
```

Output.

Going to delete an existing file

File deleted successfully!

15. GLOBAL OBJECTS

The following table provides a list of other objects which we use frequently in our applications.

Sr.No.	Module Name & Description
1	<u>Console</u> Used to print information on stdout and stderr.
2	<u>Process</u> Used to get information on current process. Provides multiple events related to process activities.

16. UTILITY MODULES

There are several utility modules available in Node.js module library. These modules are very common and are frequently used while developing any Node based application.

Sr.No.	Module Name & 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.

17. WEB MODULE

Web Server

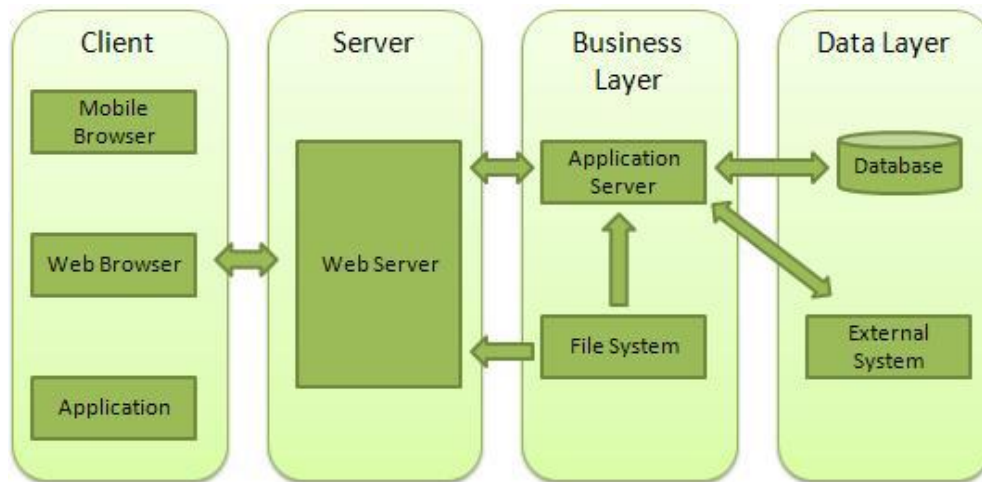
A Web Server is a software application which handles HTTP requests sent by the HTTP client, like web browsers, and returns web pages in response to the clients. Web servers usually deliver html documents along with images, style sheets, and scripts.

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.

Apache web server is one of the most commonly used web servers. It is an open source project.

Web Application Architecture

A Web application is usually divided into four layers –



- **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

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/');
```

Next let's create the following html file named index.htm in the same directory where you created server.js.

File: index.htm

```
<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
```

Output.

Server running at http://127.0.0.1:8081/

18. EXPRESS FRAMEWORK

Express Overview:

Express is a minimal and flexible Node.js web application framework that provides a robust set of features to develop web and mobile applications. It facilitates the rapid development of Node based Web applications.

Following are some of the core features of Express framework –

- Allows setting up middle wares to respond to HTTP Requests.
- Defines a routing table which is used to perform different actions based on HTTP Method and URL.
- Allows to dynamically rendering HTML Pages based on passing arguments to templates.

Installing Express

Firstly, install the Express framework globally using NPM so that it can be used to create a web application using node terminal.

```
$ npm install express --save
```

The above command saves the installation locally in the **node modules** directory and creates a directory express inside node modules. You should install the following important modules along with express –

- **body-parser** – This is a node.js middleware for handling JSON, Raw, Text and URL encoded form data.
- **cookie-parser** – Parse Cookie header and populate req.cookies with an object keyed by the cookie names.
- **multer** – This is a node.js middleware for handling multipart/form-data.

```
$ npm install body-parser --save
```

```
$ npm install cookie-parser --save
```

```
$ npm install multer --save
```

Hello world Example

Following is a very basic Express app which starts a server and listens on port 8081 for connection. This app responds with **Hello World!** for requests to the homepage. For every other path, it will respond with a **404 Not Found**.

```
var express = require('express');
var app = express();
app.get('/', function (req, res) {
  res.send('Hello World');
})
var server = app.listen(8081, function () {
  var host = server.address().address
  var port = server.address().port
  console.log("Example app listening at http://%s:%s", host, port)
})
```

Save the above code in a file named server.js and run it with the following command.

```
$ node server.js
```

Output

Example app listening at http://0.0.0.0:8081

Open <http://127.0.0.1:8081/> in any browser to see the following result.



19. RESTful API

REST stands for REpresentational State Transfer. REST is web standards based architecture and uses HTTP Protocol. It revolves around resource where every component is a resource and a resource is accessed by a common interface using HTTP standard methods. REST was first introduced by Roy Fielding in 2000.

A REST Server simply provides access to resources and REST client accesses and modifies the resources using HTTP protocol. Here each resource is identified by URIs/ global IDs. REST uses various representations to represent a resource like text, JSON, XML but JSON is the most popular one.

HTTP methods

Following four HTTP methods are commonly used in REST based architecture.

- **GET** – This is used to provide a read only access to a resource.
- **PUT** – This is used to create a new resource.
- **DELETE** – This is used to remove a resource.
- **POST** – This is used to update a existing resource or create a new resource.

RESTful Web Services

A web service is a collection of open protocols and standards used for exchanging data between applications or systems. Software applications written in various programming languages and running on various platforms can use web services to exchange data over computer networks like the Internet in a manner similar to inter-process communication on a single computer. This interoperability (e.g., communication between Java and Python, or Windows and Linux applications) is due to the use of open standards.

Web services based on REST Architecture are known as RESTful web services. These webservices uses HTTP methods to implement the concept of REST architecture. A RESTful web service usually defines a URI, Uniform Resource Identifier a service, which provides resource representation such as JSON and set of HTTP Methods.

Creating RESTful for a Library

Consider we have a JSON based database of users having the following users in a file **users.json**:

```
{
  "user1" : {
    "name" : "mahesh",
    "password" : "password1",
    "profession" : "teacher",
    "id": 1
  },

  "user2" : {
    "name" : "suresh",
    "password" : "password2",
    "profession" : "librarian",
    "id": 2
  },

  "user3" : {
    "name" : "ramesh",
    "password" : "password3",
    "profession" : "clerk",
    "id": 3
  }
}
```

Based on this information we are going to provide following RESTful APIs.

Sr.No.	URI	HTTP Method	POST body	Result
1	listUsers	GET	empty	Show list of all the users.
2	addUser	POST	JSON String	Add details of new user.
3	deleteUser	DELETE	JSON String	Delete an existing user.

4	:id	GET	empty	Show details of a user.
---	-----	-----	-------	-------------------------

List Users

Let's implement our first RESTful API **listUsers** using the following code in a server.js file –

server.js

```
var express = require('express');
var app = express();
var fs = require("fs");
app.get('/listUsers', function (req, res) {
  fs.readFile( __dirname + "/" + "users.json", 'utf8', function (err, data) {
    console.log( data );
    res.end( data );
  });
})
var server = app.listen(8081, function () {
  var host = server.address().address
  var port = server.address().port
  console.log("Example app listening at http://%s:%s", host, port)
})
```

This should produce following result –

You can change given IP address when you will put the solution in production environment.

```
"user1" :
  "name" : "mahesh",
  "password" : "password1",
  "profession" : "teacher",
  "id": 1

"user2" :
  "name" : "suresh",
  "password" : "password2",
  "profession" : "librarian",
  "id": 2

"user3" :
  "name" : "ramesh",
  "password" : "password3",
  "profession" : "clerk",
  "id": 3
```

Add User

Following API will show you how to add new user in the list. It is the detail of the new user –

```
user = {  
  "user4" : {  
    "name" : "mohit",  
    "password" : "password4",  
    "profession" : "teacher",  
    "id": 4  
  }  
}
```

Following is the **addUser** API to a new user in the database –

server.js

```
var express = require('express');  
var app = express();  
var fs = require("fs");  
  
var user = {  
  "user4" : {  
    "name" : "mohit",  
    "password" : "password4",  
    "profession" : "teacher",  
    "id": 4  
  }  
}  
  
app.post('/addUser', function (req, res) {  
  // First read existing users.  
  fs.readFile(__dirname + "/" + "users.json", 'utf8', function (err, data) {  
    data = JSON.parse( data );  
    data["user4"] = user["user4"];  
    console.log( data );  
    res.end( JSON.stringify(data));  
  });  
})  
  
var server = app.listen(8081, function () {  
  var host = server.address().address  
  var port = server.address().port  
  console.log("Example app listening at http://%s:%s", host, port)
```

```
}}
```

Output–

```
"user1":{"name":"mahesh","password":"password1","profession":"teacher","id":1},
"user2":{"name":"suresh","password":"password2","profession":"librarian","id":2},
"user3":{"name":"ramesh","password":"password3","profession":"clerk","id":3},
"user4":{"name":"mohit","password":"password4","profession":"teacher","id":4}
```

Show Detail

Now we will implement an API which will be called using user ID and it will display the detail of the corresponding user.

server.js

```
var express = require('express');
var app = express();
var fs = require("fs");

app.get('/:id', function (req, res) {
  // First read existing users.
  fs.readFile( __dirname + "/" + "users.json", 'utf8', function (err, data) {
    var users = JSON.parse( data );
    var user = users["user" + req.params.id]
    console.log( user );
    res.end( JSON.stringify(user));
  });
})

var server = app.listen(8081, function () {
  var host = server.address().address
  var port = server.address().port
  console.log("Example app listening at http://%s:%s", host, port)
})
```

This should produce following result –

```
"name":"suresh","password":"password2","profession":"librarian","id":2
```

Delete User

This API is very similar to addUser API where we receive input data through req.body and then based on user ID we delete that user from the database. To keep our program simple we assume we are going to delete user with ID 2.

server.js


```

var express = require('express');
var app = express();
var fs = require("fs");

var id = 2;

app.delete('/deleteUser', function (req, res) {
  // First read existing users.
  fs.readFile( __dirname + "/" + "users.json", 'utf8', function (err, data) {
    data = JSON.parse( data );
    delete data["user" + 2];

    console.log( data );
    res.end( JSON.stringify(data));
  });
})

var server = app.listen(8081, function () {
  var host = server.address().address
  var port = server.address().port
  console.log("Example app listening at http://%s:%s", host, port)
})

```

Output

```

{"user1":{"name":"mahesh","password":"password1","profession":"teacher","id":1},
"user3":{"name":"ramesh","password":"password3","profession":"clerk","id"}}

```

20. SCALING APPLICATION

Node.js runs in a single-thread mode, but it uses an event-driven paradigm to handle concurrency. It also facilitates creation of child processes to leverage parallel processing on multi-core CPU based systems.

Child processes always have three streams **child.stdin**, **child.stdout**, and **child.stderr** which may be shared with the stdio streams of the parent process. Node provides **child_process** module which has the following **three major ways to create a child process**.

- **exec** – `child_process.exec` method runs a command in a shell/console and buffers the output.
- **spawn** – `child_process.spawn` launches a new process with a given command.
- **fork** – The `child_process.fork` method is a special case of the `spawn()` to create child processes.

The exec() method

child_process.exec method runs a command in a shell and buffers the output. It has the following signature –

child_process.exec(command[, options], callback)

Parameters

Here is the description of the parameters used –

- **command** (String) The command to run, with space-separated arguments
- **options** (Object) may comprise one or more of the following options –
 - **cwd** (String) Current working directory of the child process
 - **env** (Object) Environment key-value pairs
 - **encoding** (String) (Default: 'utf8')
 - **shell** (String) Shell to execute the command with (Default: '/bin/sh' on UNIX, 'cmd.exe' on Windows, The shell should understand the -c switch on UNIX or /s /c on Windows. On Windows, command line parsing should be compatible with cmd.exe.)
 - **timeout** (Number) (Default: 0)
 - **maxBuffer** (Number) (Default: 200*1024)
 - **killSignal** (String) (Default: 'SIGTERM')
 - **uid** (Number) Sets the user identity of the process.
 - **gid** (Number) Sets the group identity of the process.
- **callback** The function gets three arguments **error**, **stdout**, and **stderr** which are called with the output when the process terminates.

The **exec() method** returns a buffer with a max size and waits for the process to end and tries to return all the buffered data at once.

Example

Let us create two js files named support.js and master.js –

File: support.js

```
console.log("Child Process " + process.argv[2] + " executed.");
```

File: master.js

```
const fs = require('fs');
const child_process = require('child_process');

for(var i=0; i<3; i++) {
```

```
var workerProcess = child_process.exec('node support.js '+i,function
(error, stdout, stderr) {

    if (error) {
        console.log(error.stack);
        console.log('Error code: '+error.code);
        console.log('Signal received: '+error.signal);
    }
    console.log('stdout: ' + stdout);
    console.log('stderr: ' + stderr);
});

workerProcess.on('exit', function (code) {
    console.log('Child process exited with exit code '+code);
});
}
```

Output.

Server has started.

Child process exited with exit code 0

stdout: Child Process 1 executed.

stderr:

Child process exited with exit code 0

stdout: Child Process 0 executed.

stderr:

Child process exited with exit code 0

stdout: Child Process 2 executed.

The spawn() Method

child_process.spawn method launches a new process with a given command. It has the following signature –

child_process.spawn(command[, args][, options])

Parameters

Here is the description of the parameters used –

- **command** (String) The command to run

- **args** (Array) List of string arguments
- **options** (Object) may comprise one or more of the following options –
 - **cwd** (String) Current working directory of the child process.
 - **env** (Object) Environment key-value pairs.
 - **stdio** (Array) String Child's stdio configuration.
 - **customFds** (Array) Deprecated File descriptors for the child to use for stdio.
 - **detached** (Boolean) The child will be a process group leader.
 - **uid** (Number) Sets the user identity of the process.
 - **gid** (Number) Sets the group identity of the process.

The `spawn()` method returns streams (`stdout` & `stderr`) and it should be used when the process returns a volume amount of data. `spawn()` starts receiving the response as soon as the process starts executing.

Example

```
const fs = require('fs');
const child_process = require('child_process');

for(var i = 0; i<3; i++) {
  var workerProcess = child_process.spawn('node', ['support.js', i]);

  workerProcess.stdout.on('data', function (data) {
    console.log('stdout: ' + data);
  });

  workerProcess.stderr.on('data', function (data) {
    console.log('stderr: ' + data);
  });

  workerProcess.on('close', function (code) {
    console.log('child process exited with code ' + code);
  });
}
```

Output. Server has started

stdout: Child Process 0 executed.

child process exited with code 0

stdout: Child Process 1 executed.

stdout: Child Process 2 executed.

child process exited with code 0

child process exited with code 0

The fork() Method

child_process.fork method is a special case of spawn() to create Node processes. It has the following signature –

child_process.fork(modulePath[, args][, options])

Parameters

Here is the description of the parameters used –

- **modulePath** (String) The module to run in the child.
- **args** (Array) List of string arguments
- **options** (Object) may comprise one or more of the following options –
 - **cwd** (String) Current working directory of the child process.
 - **env** (Object) Environment key-value pairs.
 - **execPath** (String) Executable used to create the child process.
 - **execArgv** (Array) List of string arguments passed to the executable (Default: process.execArgv).
 - **silent** (Boolean) If true, stdin, stdout, and stderr of the child will be piped to the parent, otherwise they will be inherited from the parent, see the "pipe" and "inherit" options for spawn()'s stdio for more details (default is false).
 - **uid** (Number) Sets the user identity of the process.
 - **gid** (Number) Sets the group identity of the process.

The fork method returns an object with a built-in communication channel in addition to having all the methods in a normal ChildProcess instance.

Example

Create two js files named support.js and master.js –

File: support.js

```
console.log("Child Process " + process.argv[2] + " executed.");
```

File: master.js

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

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

for(var i=0; i<3; i++) {
  var worker_process = child_process.fork("support.js", [i]);

  worker_process.on('close', function (code) {
    console.log('child process exited with code ' + code);
  });
}
```

Output. Server has started.

Child Process 0 executed.

Child Process 1 executed.

Child Process 2 executed.

child process exited with code 0

child process exited with code 0

child process exited with code 0

21. PACKAGING

JXcore, which is an open source project, introduces a unique feature for packaging and encryption of source files and other assets into JX packages.

Consider you have a large project consisting of many files. JXcore can pack them all into a single file to simplify the distribution. This chapter provides a quick overview of the whole process starting from installing JXcore.

JXcore Installation

Installing JXcore is quite simple. Here we have provided step-by-step instructions on how to install JXcore on your system.

Follow the steps given below –

Step 1

Download the JXcore package from <https://github.com/jxcore/jxcore>, as per your operating system and machine architecture. We downloaded a package for Cenots running on 64-bit machine.

```
$ wget https://s3.amazonaws.com/nodejx/jx_rh64.zip
```

Step 2

Unpack the downloaded file **jx_rh64.zip** and copy the jx binary into /usr/bin or may be in any other directory based on your system setup.

```
$ unzip jx_rh64.zip
```

```
$ cp jx_rh64/jx /usr/bin
```

Step 3

Set your PATH variable appropriately to run jx from anywhere you like.

```
$ export PATH=$PATH:/usr/bin
```

Step 4

You can verify your installation by issuing a simple command as shown below. You should find it working and printing its version number as follows –

```
$ jx --version
```

```
v0.10.32
```

Packaging the Code

Consider you have a project with the following directories where you kept all your files including Node.js, main file, index.js, and all the modules installed locally.

```
drwxr-xr-x  2 root root  4096 Nov 13 12:42 images
-rwxr-xr-x  1 root root 30457 Mar  6 12:19 index.htm
-rwxr-xr-x  1 root root 30452 Mar  1 12:54 index.js
drwxr-xr-x 23 root root  4096 Jan 15 03:48 node_modules
drwxr-xr-x  2 root root  4096 Mar 21 06:10 scripts
drwxr-xr-x  2 root root  4096 Feb 15 11:56 style
```

To package the above project, you simply need to go inside this directory and issue the following jx command. Assuming index.js is the entry file for your Node.js project –

```
$ jx package index.js index
```

Here you could have used any other package name instead of **index**. We have used **index** because we wanted to keep our main file name as index.jx. However, the above command will pack everything and will create the following two files –

- **index.jxp** This is an intermediate file which contains the complete project detail needed to compile the project.
- **index.jx** This is the binary file having the complete package that is ready to be shipped to your client or to your production environment.

Launching JX File

Consider your original Node.js project was running as follows –

```
$ node index.js command_line_arguments
```

After compiling your package using JXcore, it can be started as follows –

```
$ jx index.jx command_line_arguments
```

To know more on JXcore, you can check its official website.

~~~ End of the Unit –II ~~~