Rakesh Kumar

Abstract

This Notes have been referred to various tutorials and documents present there online

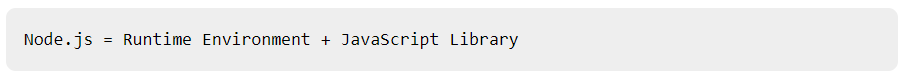
NODE JS

NOTEBOOK

**What is Node.js?**

**Node.js is a server-side platform** built on Google Chrome's JavaScript Engine (V8 Engine). Node.js runs single-threaded, non-blocking, asynchronous programming, which is very memory efficient.

* **easily building fast** and **scalable network and server-side applications.**
* **uses an event-driven, non-blocking I/O model** that makes it lightweight and efficient**, perfect for data-intensive** (not CPU intensive like File conversion / compression, Image Manipuation, etc.) real-time applications that run across distributed devices.
* is an open source, **cross-platform runtime environment** for developing server-side and networking applications.
* also provides a rich library of various JavaScript modules.



## Features of Node.js

* **Asynchronous and Event Driven: -** that is, non-blocking. It essentially means a Node.js based server never waits for an API to return data. Node.js eliminates the waiting, and simply continues with the next request.
* **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**: - 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 .
* **No Buffering**: - Node.js applications never buffer any data. These applications simply output the data in chunks.
* **Open source**.

## Who Uses Node.js?

## This list includes eBay, General Electric, GoDaddy, Microsoft, PayPal, Uber, Wikipins, Yahoo!, and Yammer to name a few.

## 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

## Where Not to Use Node.js?

It is not advisable to use Node.js for CPU intensive applications operations like image processing or other heavy computation work because it takes time to process a request and thereby blocks the single thread.

**Node.js - First Application**

There can be console-based and web-based node.js applications.

**Verify installation: Executing a File**

Create a js file named **main.js** on your machine (Windows or Linux) having the following code.

/\* Hello, World! program in node.js \*/

console.log("Hello, World!")

Now execute main.js file using Node.js interpreter to see the result −

$ node main.js

If everything is fine with your installation, this should produce the following result −

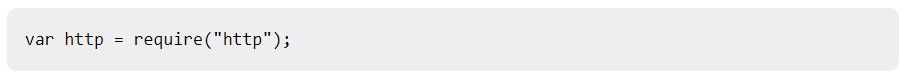
Hello, World!

## Node.js web-based Example

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

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

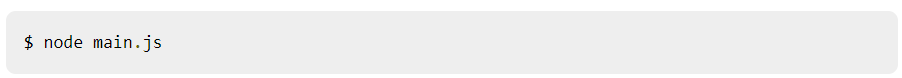
We use the **require** directive to load the http module and store the returned HTTP instance into an http variable as follows –

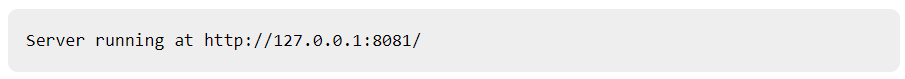


* **Create server** − A server which will listen to client's requests similar to Apache HTTP Server. We use the created http instance and call **http.createServer()** method to create a server instance and then we bind it at port 8081 using the **listen** method associated with the server instance.
* **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.



Now execute the main.js to start the server as follows –

  
Verify the Output. Server has started.

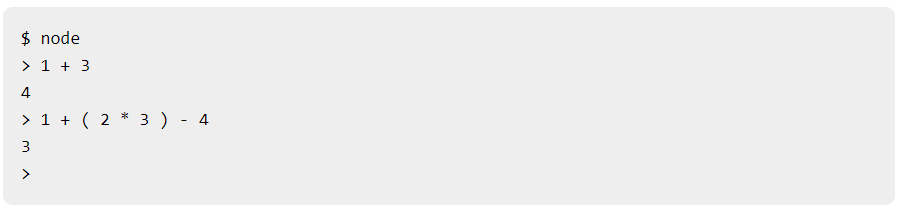


# **Node.js - REPL Terminal**

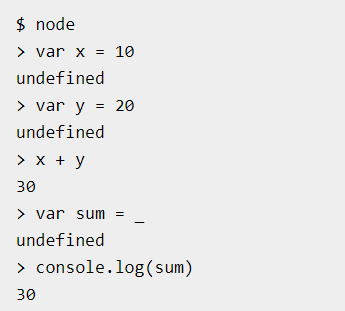
REPL stands for Read Eval Print Loop and it represents a computer environment like a Windows console

Node.js or **Node** comes bundled with a REPL environment. It performs the following tasks −

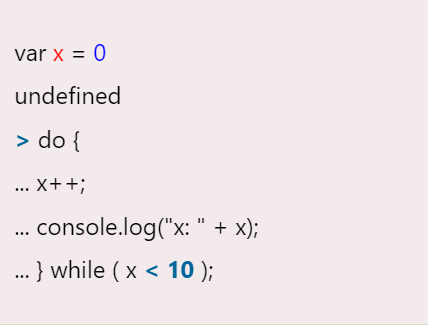
* **Read** − Reads user's input, parses the input into JavaScript data-structure, and stores in memory.
* **Eval** − Takes and evaluates the data structure.
* **Print** − Prints the result.
* **Loop** − Loops the above command until the user presses **ctrl-c** twice.

REPL can be started by simply running **node** on shell/console without any arguments .Example:-  


* You can use underscore **(\_)** to get the last result.



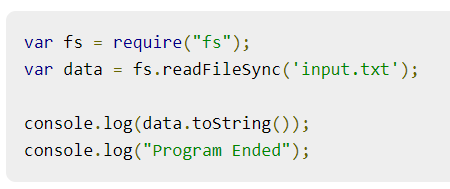
* Node REPL supports multiline expression like in JavaScript.



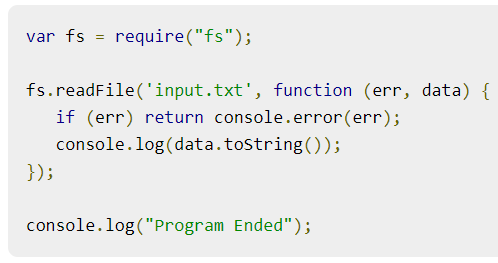
## What is Callback?

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 call-backs. All the APIs of Node are written in such a way that they support call-backs.

**TYPES**:-

* **SYNCHRONOUS / BLOCKING CODE: -**shows that the program blocks until it reads the file and then only it proceeds to end the program. Thus, a blocking program executes very much in sequence.  
  
* **ASYNCHRONOUS / NON-BLOCKING CODE: -**

shows that the program does not wait for file reading and proceeds to print "Program Ended" and at the same time, the program without blocking continues reading the file.

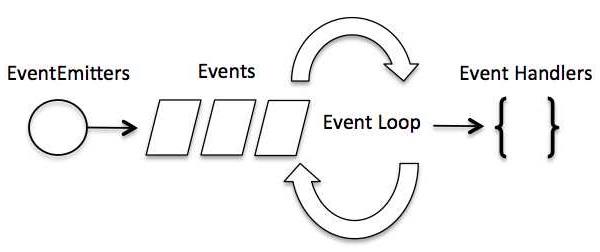


# **Node.js - Event Loop**

\*\* 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

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.

  
// 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.");

IT should produce the following result −

connection successful.

data received successfully.

Program Ended.

## How Node Applications Work?

In Node Application, any async function accepts a callback as the last parameter and a callback function accepts an error as the first parameter.

var fs = require("fs");

fs.readFile('input.txt', function (err, data) {

if (err) {

console.log(err.stack);

return;

}

console.log(data.toString());

});

console.log("Program Ended");

# **Node.js - Event Emitter**

Many objects in a Node emit events,

// 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:-

## addListener(event, listener):- Adds a listener at the end of the listeners array for the specified event.

## on(event, listener):- Adds a listener at the end of the listeners array for the specified event.

## 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.

## removeListener(event, listener):- Removes a listener from the listener array for the specified event.

## removeAllListeners([event]):- Removes all listeners, or those of the specified event.

## setMaxListeners(n):- By default, EventEmitters will print a warning if more than 10 listeners are added for a particular event.

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

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

## Class Methods

## listenerCount(emitter, event) -- > Returns the number of listeners for a given event.

## Events

**newListener**

* **event** − String: the event name
* **listener** − Function: the event handler function

**removeListener**

* **event** − String The event name
* **listener** − Function The event handler function

var events = require('events');

var eventEmitter = new events.EventEmitter();

// listener #1

var listner1 = function listner1() {

console.log('listner1 executed.');

}

// listener #2

var listner2 = function listner2() {

console.log('listner2 executed.');

}

// Bind the connection event with the listner1 function

eventEmitter.addListener('connection', listner1);

// Bind the connection event with the listner2 function

eventEmitter.on('connection', listner2);

var eventListeners = require('events').EventEmitter.listenerCount

(eventEmitter,'connection');

console.log(eventListeners + " Listner(s) listening to connection event");

// Fire the connection event

eventEmitter.emit('connection');

// Remove the binding of listner1 function

eventEmitter.removeListener('connection', listner1);

console.log("Listner1 will not listen now.");

// Fire the connection event

eventEmitter.emit('connection');

eventListeners = require('events').EventEmitter.listenerCount(eventEmitter,'connection');

console.log(eventListeners + " Listner(s) listening to connection event");

console.log("Program Ended.");

Now run the main.js to see the result −

$ node main.js

Verify the Output.

2 Listner(s) listening to connection event

listner1 executed.

listner2 executed.

Listner1 will not listen now.

listner2 executed.

1 Listner(s) listening to connection event

Program Ended.

# **Node.js – Buffers**

**The Buffer class in Node. js is used to perform operations on raw binary data.** The buffers module provides a way of handling streams of binary data. The Buffer object is a global object in Node.js, and it is not necessary to import it using the require keyword.

\*\*Generally, Buffer refers to the particular memory location in memory. Buffer and array have some similarities, but the difference is array can be any type, and it can be resizable. Buffer is mainly used to store binary data, while reading from a file or receiving packets over the network.

**The syntax for creating an empty Buffer of the length 15:**

var buf = Buffer.alloc(15);

**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")

**Following is the syntax to create a Buffer from a given array −**

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

buf = new Buffer(256);

len = buf.write("Simply Easy Learning");

console.log("Octets written : "+ len);

O/P 🡺 Octets written : 20

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

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

var buf = new Buffer('Simply Easy Learning');

var json = buf.toJSON(buf);

console.log(json);

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

var buffer1 = new Buffer('TutorialsPoint ');

var buffer2 = new Buffer('Simply Easy Learning');

var buffer3 = Buffer.concat([buffer1,buffer2]);

console.log("buffer3 content: " + buffer3.toString());

buffer3 content: TutorialsPoint Simply Easy Learning

//Compare Buffer  
var result = buffer1.compare(buffer2);

//Copy Buffer

buffer1.copy(buffer2);

//slicing a buffer

var buffer2 = buffer1.slice(0,9);

//length of the buffer

console.log("buffer length: " + buffer.length);

# **Node.js - Streams**

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

Each type of Stream is an **EventEmitter** instance and throws several events at different instance of times. For example, some of the commonly used events are −

* **data** − This event is fired when there is data is available to read.
* **end** − This event is fired when there is no more data to read.
* **error** − This event is fired when there is any error receiving or writing data.
* **finish** − This event is fired when all the data has been flushed to underlying system.

## Reading from a Stream

var fs = require("fs");

var data = '';

var data 2= 'Simply Easy Learning';

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*READING STREAM\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// 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);

});

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*WRITING STREAM\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
// 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");

## Piping the Streams

Piping is a mechanism where we provide the output of one stream as the input to another stream.

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");

## 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.

var fs = require("fs");

var zlib = require('zlib');

// Compress the file input.txt to input.txt.gz

fs.createReadStream('input.txt') // fs.createReadStream('input.txt.gz') while decompress

.pipe(zlib.createGzip()) // To unzip (decompress) .pipe(zlib.createGunzip())

.pipe(fs.createWriteStream('input.txt.gz'));

console.log("File Compressed.");

# **Node.js - File System**

Node implements File I/O using simple wrappers around standard POSIX functions. The Node File System (fs) module can be imported using the following syntax −

var fs = require("fs")

Common use for the File System module:

* Read files
* Create files
  + - Fs.appendFile()

fs.appendFile('mynewfile1.txt', 'Hello content!', function (err) {

// will append Hello content to the end of the file content present already

if (err) throw err;

console.log('Saved!');

});

* + - Fs.open()
    - Fs.writeFile()

The fs.writeFile() method replaces the specified file and content:

* Update files
  + - fs.appendFile()
    - Fs.writeFile()
* Delete files

The fs.unlink() method deletes the specified file:

* Rename File

The fs.rename() method renames the specified file:

\*\*Every method in the fs module has **synchronous** as well as **asynchronous** forms.

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");

**Verify the Output.**

**Synchronous read:** Tutorials Point is giving self learning content

to teach the world in simple and easy way!!!!!

**Program Ended**

**Asynchronous read**: Tutorials Point is giving self learning content

to teach the world in simple and easy way!!!!!

## Open a File

fs.open('input.txt', 'r+', function(err, fd) {

if (err) {

return console.error(err);

}

console.log("File opened successfully!");

});

## Writing a File

fs.writeFile('input.txt', 'Simply Easy Learning!', function(err) {

if (err) { return console.error(err); }

});

## Reading a 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());

}

});

});

## Closing a File

// Close the opened file.

fs.close(fd, function(err) {

if (err) {

console.log(err);

}

console.log("File closed successfully.");

});

## Delete a File

fs.unlink('input.txt', function(err) {

if (err) {

return console.error(err);

}

console.log("File deleted successfully!");

});

## Create a Directory

fs.mkdir('/tmp/test',function(err) {

if (err) {

return console.error(err);

}

console.log("Directory created successfully!");

});

## Read a Directory

fs.readdir("/tmp/",function(err, files) {

if (err) {

return console.error(err);

}

files.forEach( function (file) {

console.log( file );

});

});

## Remove a Directory

fs.rmdir("/tmp/test",function(err) {

if (err) {

return console.error(err);

}

console.log("Going to read directory /tmp");

});

# **Node.js - Global Objects**

Node.js global objects are global in nature and they are available in all modules. We do not need to include these objects in our application, rather we can use them directly. These objects are modules, functions, strings and object itself as explained below.

## \_\_filename

// Let's try to print the value of \_\_filename

console.log( \_\_filename );

Based on the location of your program, it will print the main file name as follows −

/web/com/1427091028\_21099/main.js

## \_\_dirname

console.log( \_\_dirname );

Based on the location of your program, it will print current directory name as follows –

/web/com/1427091028\_21099

## setTimeout(cb, ms)

## clearTimeout(t)

## setInterval(cb, ms)

# **Node.js - Utility Modules**

|  |  |
| --- | --- |
| **Sr.No.** | **Module Name & Description** |
| 1 | [OS Module](https://www.tutorialspoint.com/nodejs/nodejs_os_module.htm):-Provides basic operating-system related utility functions. |
| 2 | [Path Module](https://www.tutorialspoint.com/nodejs/nodejs_path_module.htm):- Provides utilities for handling and transforming file paths. |
| 3 | [Net Module](https://www.tutorialspoint.com/nodejs/nodejs_net_module.htm):-Provides both servers and clients as streams. Acts as a network wrapper. |
| 4 | [DNS Module](https://www.tutorialspoint.com/nodejs/nodejs_dns_module.htm):- Provides functions to do actual DNS lookup as well as to use underlying operating system name resolution functionalities. |
| 5 | [Domain Module](https://www.tutorialspoint.com/nodejs/nodejs_domain_module.htm) :- Provides ways to handle multiple different I/O operations as a single group. |

# **Node.js - Web Module**

## What is a Web Server?

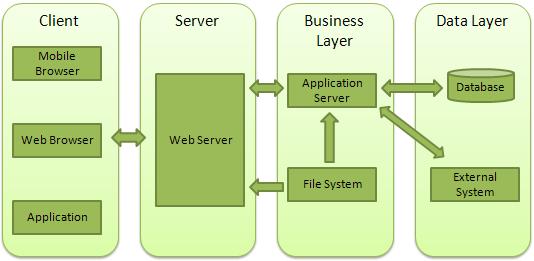
A Web Server is a software application which handles HTTP requests sent by the HTTP client, like web browsers, and returns web pages in response to the clients. Web servers usually deliver html documents along with images, style sheets, and scripts.

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.

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

**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'}); //If the response from the HTTP server is supposed to be displayed as HTML, you should include an HTTP header with the correct content type:

} else {

//Page found // HTTP Status: 200 : OK // Content Type: text/plain

response.writeHead(200, {'Content-Type': 'text/html'});

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

}

response.end(); // Send the response body

});

}).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>

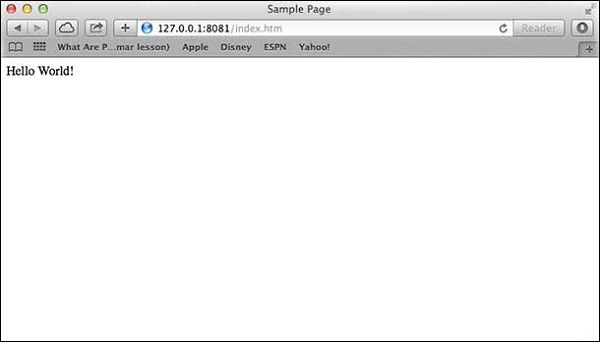
</head>

<body>

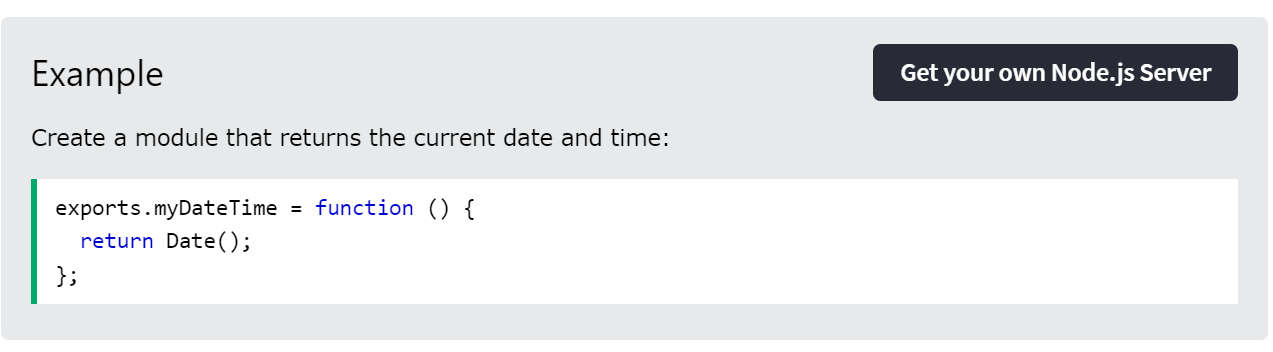
Hello World!

</body>

</html>

Open http://127.0.0.1:8081/index.htm in any browser to see the following result. 

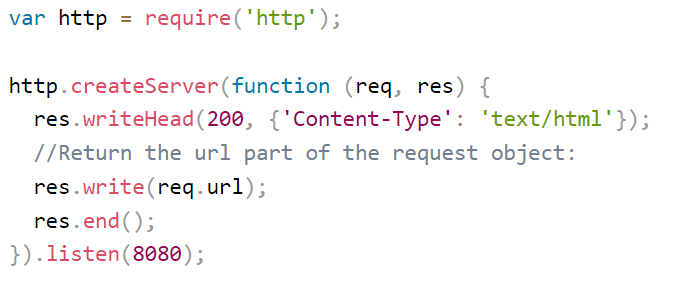
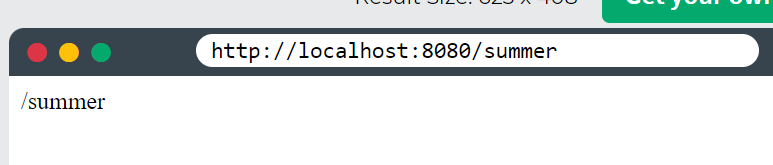
**Create Your Own Modules**

****

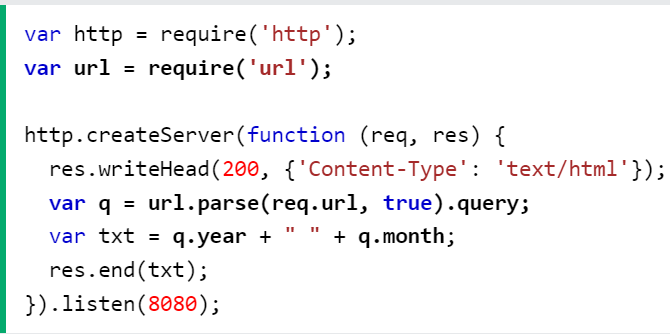
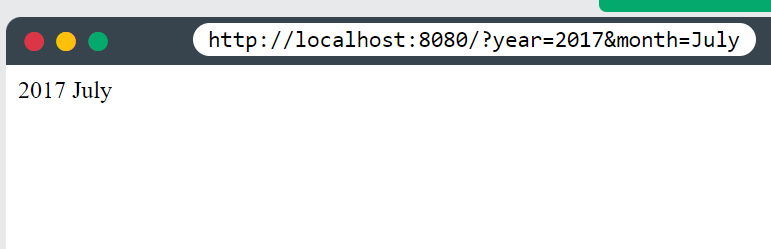
Use the exports keyword to make properties and methods available outside the module file. Now you can include and use the module in any of your Node.js files.  
ex:- var dt = require('./myfirstmodule');

**The Built-in URL Module:-**

**Read the Query String**



**Split the Query String**



**Node.js Upload Files**

The Formidable Module

There is a very good module for working with file uploads, called "Formidable".

**Step 1: Create an Upload Form**

**Step 2: Parse the Uploaded File**

**Step 3: Save the File**

**Example:-**

var http = require('http');

var formidable = require('formidable');

var fs = require('fs');

http.createServer(function (req, res) {

if (req.url == '/fileupload') {

var form = new formidable.IncomingForm();

form.parse(req, function (err, fields, files) {

var oldpath = files.filetoupload.filepath;

var newpath = 'C:/Users/Your Name/' + files.filetoupload.originalFilename;

fs.rename(oldpath, newpath, function (err) {

if (err) throw err;

res.write('File uploaded and moved!');

res.end();

});

});

} else {

res.writeHead(200, {'Content-Type': 'text/html'});

res.write('<form action="fileupload" method="post" enctype="multipart/form-data">');

res.write('<input type="file" name="filetoupload"><br>');

res.write('<input type="submit">');

res.write('</form>');

return res.end();

}

}).listen(8080);

**Node.js Send an Email**

The **Nodemailer** module makes it easy to send emails from your computer.

var nodemailer = require('nodemailer');

var transporter = nodemailer.createTransport({

service: 'gmail',

auth: {

user: 'youremail@gmail.com',

pass: 'yourpassword'

}

});

var mailOptions = {

from: 'youremail@gmail.com',

to: 'myfriend@yahoo.com, [myotherfriend@yahoo.com](mailto:myotherfriend@yahoo.com)',

subject: 'Sending Email using Node.js',

text: 'That was easy!',

html: '<h1>Welcome</h1><p>That was easy!</p>'

};

transporter.sendMail(mailOptions, function(error, info){

if (error) { console.log(error); }

else { console.log('Email sent: ' + info.response); }

});

**Access Global Scope**

In a browser, global scope is the window object. In Node.js, global object represents the global scope.

To add something in global scope, you need to export it using export or module.export. The same way, import modules/object using require() function to access it from the global scope.

For example, to export an object in Node.js, use exports.name = object.

Example:

exports.log = {

console: function(msg) {

console.log(msg);

},

file: function(msg) {

// log to file here

}

}

Now, you can import log object using require() function and use it anywhere in your Node.js project.

**Node.js Built-in Modules**

<https://www.w3schools.com/nodejs/ref_modules.asp>

Node.js includes 3 types of modules:

* **Core Modules** 🡪 http , fs, url, path, queryString, e.t.c
* **Local Modules** 🡪 which user creates to access it locally in that project

**Example**:-

var bannerConsole = {

info: function (info) {

console.log('Info: ' + info);

},

warning:function (warning) {

console.log('Warning: ' + warning);

},

error:function (error) {

console.log('Error: ' + error);

}

};

module.exports = bannerConsole

Use **module.exports** or **exports** to expose a function, object or variable as a module in Node.js. The module is a variable that represents the current module, and exports is an object that will be exposed as a module. So, whatever you assign to module.exports will be exposed as a module.

Ex:-

exports.SimpleMessage = 'Hello world';

//OR

module.exports.SimpleMessage = 'Hello world';

//OR

module.exports = 'Hello world';

**USING LOCAL MODULES**:-

var myLogModule = require('./Log.js');

myLogModule.info('Node.js started');

* **Third Party Modules** 🡪 user downloads and use using NPM install

**Debug Node.js Application**

You can debug Node.js application using various tools including following:

* **Core Node.js debugger**

var fs = require('fs');

fs.readFile('test.txt', 'utf8', function (err, data) {

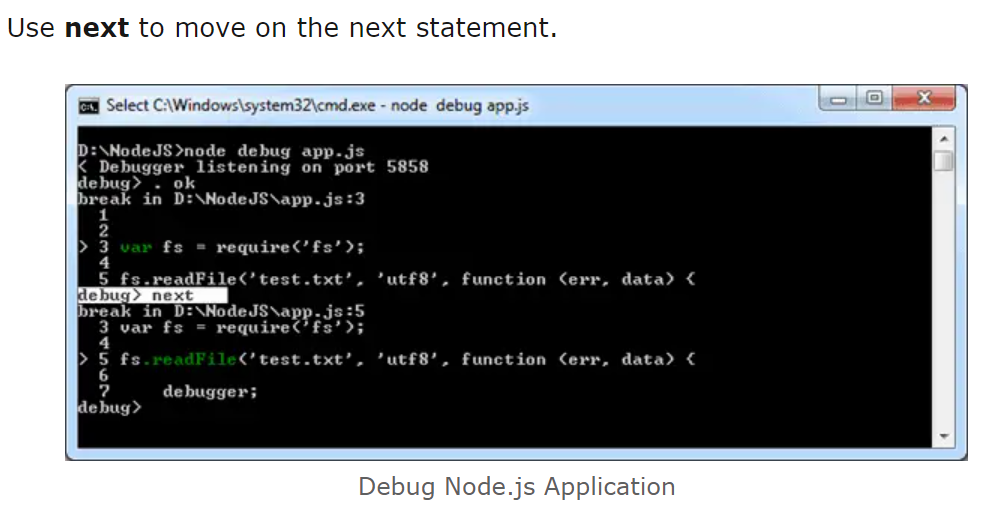
**debugger**;

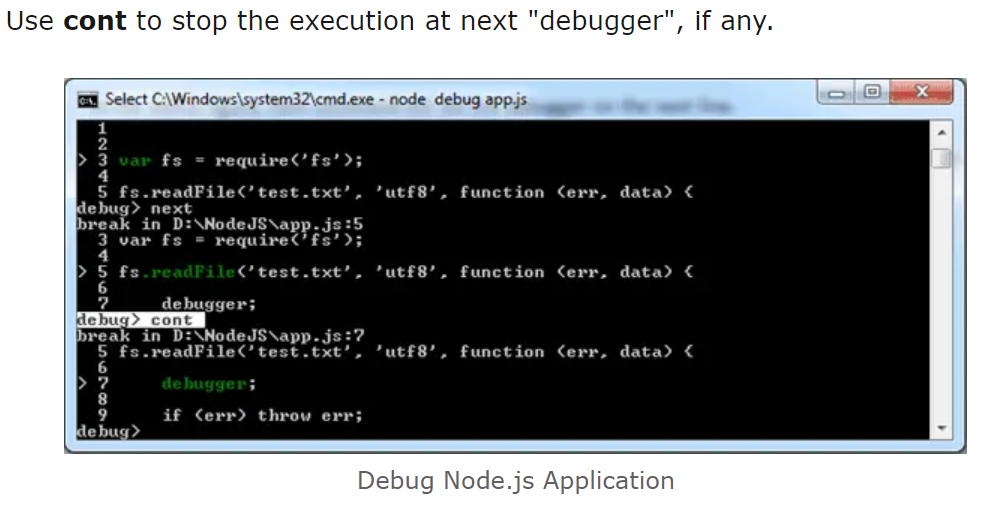
if (err) throw err;

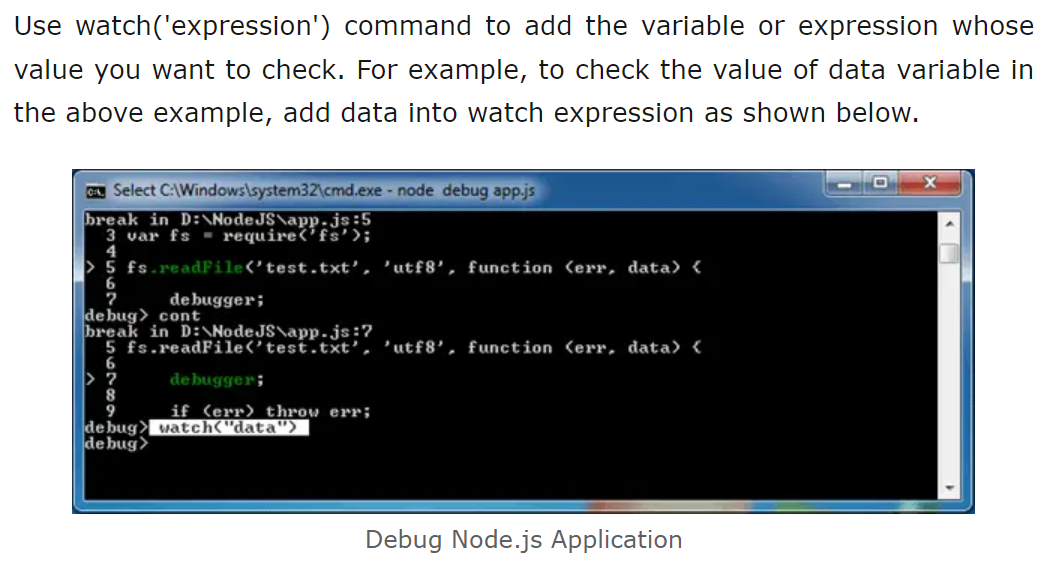
console.log(data);

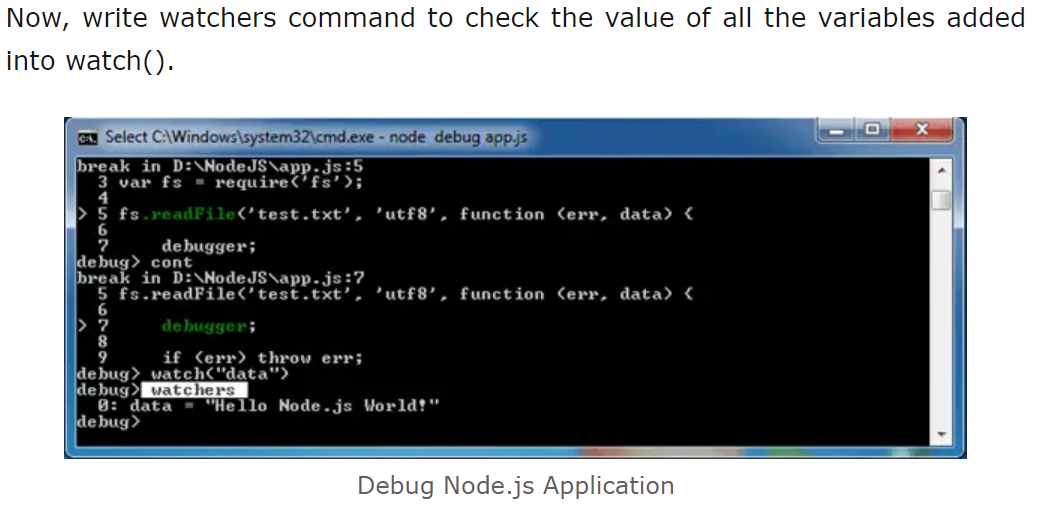
});











* **Node Inspector**
* **Built-in debugger in IDEs**