# Node.js Tutorials

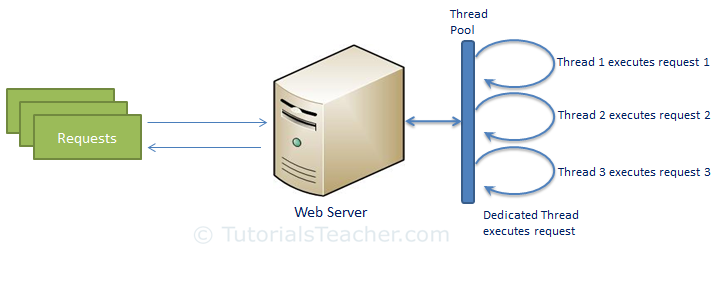
Node.js is an open-source server side runtime environment built on Chrome's V8 JavaScript engine. It provides an event driven, non-blocking (asynchronous) I/O and cross-platform runtime environment for building highly scalable server-side applications using JavaScript.

# Node.js Process Model

In this section, we will learn about the Node.js process model and understand why we should use Node.js.

## 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.

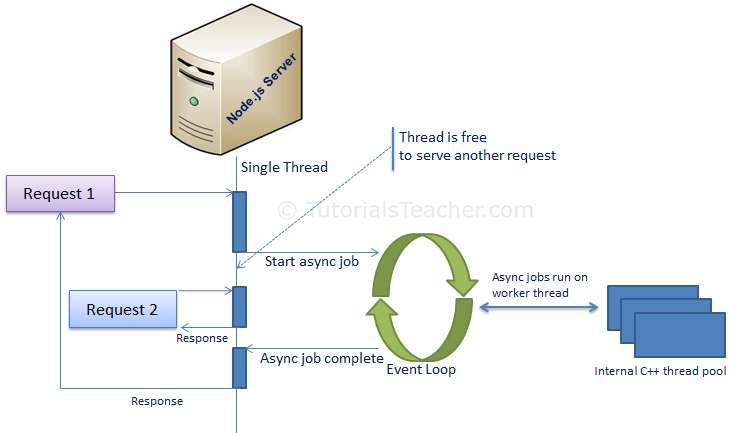
[](http://www.tutorialsteacher.com/Content/images/nodejs/traditional-web-server-model.png)Traditional Web Server Model

## 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.

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](http://software.schmorp.de/pkg/libev.html) for the event loop which in turn uses internal C++ thread pool to provide asynchronous I/O.

The following figure illustrates asynchronous web server model using Node.js.

[](http://www.tutorialsteacher.com/Content/images/nodejs/nodejs-process-model.png)Node.js Process Model

Node.js process model increases the performance and scalability with a few caveats. Node.js is not fit for an application which performs CPU-intensive operations like image processing or other heavy computation work because it takes time to process a request and thereby blocks the single thread.

Setup Node.js Development Environment

In this section, you will learn about the tools required and steps to setup development environment to develop a Node.js application.

Node.js development environment can be setup in Windows, Mac, Linux and Solaris. The following tools/SDK are required for developing a Node.js application on any platform.

1. Node.js
2. Node Package Manager (NPM)
3. IDE (Integrated Development Environment) or TextEditor

NPM (Node Package Manager) is included in Node.js installation since Node version 0.6.0., so there is no need to install it separately.

After installation, verify the Node.js installation using terminal window and enter the following command. It will display the version number of Node.js installed on your Mac.

$ node -v

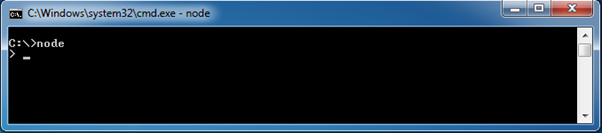
Optionally, for Mac or Linux users, you can directly install Node.js from the command line using Homebrew package manager for Mac OS or Linuxbrew package manager for Linux Operating System. For Linux, you will need to install additional dependencies, viz. Ruby version 1.8.6 or higher and GCC version 4.2 or higher before installing node.

$ brew install node

# Node.js Console - REPL

Node.js comes with virtual environment called REPL (aka Node shell). REPL stands for Read-Eval-Print-Loop. It is a quick and easy way to test simple Node.js/JavaScript code.

To launch the REPL (Node shell), open command prompt (in Windows) or terminal (in Mac or UNIX/Linux) and type *node* as shown below. It will change the prompt to > in Windows and MAC.

[](http://www.tutorialsteacher.com/Content/images/nodejs/node-repl.png)Launch Node.js REPL

You can now test pretty much any Node.js/JavaScript expression in REPL. For example, if your write "10 + 20" then it will display result 30 immediately in new line.

> 10 + 20   
30

The + operator also concatenates strings as in browser's JavaScript.

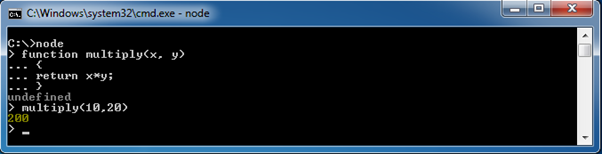
> "Hello" + "World"   
Hello World

You can also define variables and perform some operation on them.

> var x = 10, y = 20;   
> x + y   
30

If you need to write multi line JavaScript expression or function then just press **Enter** whenever you want to write something in the next line as a continuation of your code. The REPL terminal will display three dots (...), it means you can continue on next line. Write .break to get out of continuity mode.

For example, you can define a function and execute it as shown below.

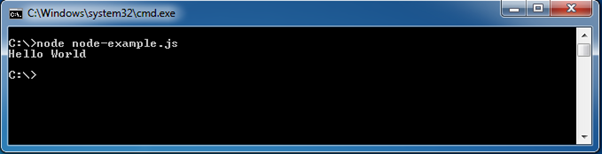
[](http://www.tutorialsteacher.com/Content/images/nodejs/nodejs-example2.png)Node.js Example in REPL

You can execute an external JavaScript file by writing node fileName command. For example, assume that node-example.js is on C drive of your PC with following code.

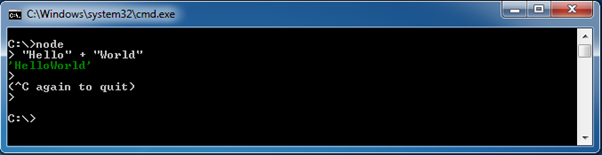
node-example.js

console.log("Hello World");

Now, you can execute node-exampel.js from command prompt as shown below.

[](http://www.tutorialsteacher.com/Content/images/nodejs/run-nodejs-external-file.png)Run External JavaScript file

To exit from the REPL terminal, press Ctrl + C twice or write .exit and press Enter.

[](http://www.tutorialsteacher.com/Content/images/nodejs/quit-repl.png)Quit from REPL

Thus, you can execute any Node.js/JavaScript code in the node shell (REPL). This will give you a result which is similar to the one you will get in the console of Google Chrome browser.

Note: ECMAScript implementation in Node.js and browsers is slightly different. For example, {}+{} is '[object Object][object Object]' in Node.js REPL, whereas the same code is NaN in the Chrome console because of the automatic semicolon insertion feature. However, mostly Node.js REPL and the Chrome/Firefox consoles are similar.

The following table lists important REPL commands.

| REPL Command | Description |
| --- | --- |
| .help | Display help on all the commands |
| tab Keys | Display the list of all commands. |
| Up/Down Keys | See previous commands applied in REPL. |
| .save filename | Save current Node REPL session to a file. |
| .load filename | Load the specified file in the current Node REPL session. |
| ctrl + c | Terminate the current command. |
| ctrl + c (twice) | Exit from the REPL. |
| ctrl + d | Exit from the REPL. |
| .break | Exit from multiline expression. |
| .clear | Exit from multiline expression. |

Node.js Basics

Node.js supports JavaScript. So, JavaScript syntax on Node.js is similar to the browser's JavaScript syntax.

Visit [JavaScript](http://www.tutorialsteacher.com/javascirpt) section to learn about JavaScript syntax in detail.

Primitive Types

Node.js includes following primitive types:

* String
* Number
* Boolean
* Undefined
* Null
* RegExp

Everything else is an object in Node.js.

Loose Typing

JavaScript in Node.js supports loose typing like the browser's JavaScript. Use var keyword to declare a variable of any type.

Object Literal

Object literal syntax is same as browser's JavaScript.

Example: Object

var obj = {

authorName: 'Ryan Dahl',

language: 'Node.js'

}

Functions

Functions are first class citizens in Node's JavaScript, similar to the browser's JavaScript. A function can have attributes and properties also. It can be treated like a class in JavaScript.

Example: Function

function Display(x) {

console.log(x);

}

Display(100);

Buffer

Node.js includes an additional data type called Buffer (not available in browser's JavaScript). Buffer is mainly used to store binary data, while reading from a file or receiving packets over the network.

process object

Each Node.js script runs in a process. It includes **process** object to get all the information about the current process of Node.js application.

The following example shows how to get process information in REPL using **process** object.

> process.execPath   
'C:\\Program Files\\nodejs\\node.exe'  
> process.pid  
1652  
> process.cwd()  
'C:\\'

Defaults to local

Node's JavaScript is different from browser's JavaScript when it comes to global scope. In the browser's JavaScript, variables declared without var keyword become global. In Node.js, everything becomes local by default.

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

}

}

# Node.js Module

Module in Node.js is a simple or complex functionality organized in single or multiple JavaScript files which can be reused throughout the Node.js application.

Each module in Node.js has its own context, so it cannot interfere with other modules or pollute global scope. Also, each module can be placed in a separate .js file under a separate folder.

Node.js implements [CommonJS modules standard](http://requirejs.org/docs/commonjs.html" \t "_blank). CommonJS is a group of volunteers who define JavaScript standards for web server, desktop, and console application.

## Node.js Module Types

Node.js includes three types of modules:

1. Core Modules
2. Local Modules
3. Third Party Modules

## Node.js Core Modules

Node.js is a light weight framework. The core modules include bare minimum functionalities of Node.js. These core modules are compiled into its binary distribution and load automatically when Node.js process starts. However, you need to import the core module first in order to use it in your application.

The following table lists some of the important core modules in Node.js.

| Core Module | Description |
| --- | --- |
| [http](https://nodejs.org/api/http.html) | http module includes classes, methods and events to create Node.js http server. |
| [url](https://nodejs.org/api/url.html) | url module includes methods for URL resolution and parsing. |
| [querystring](https://nodejs.org/api/querystring.html) | querystring module includes methods to deal with query string. |
| [path](https://nodejs.org/api/path.html) | path module includes methods to deal with file paths. |
| [fs](https://nodejs.org/api/fs.html) | fs module includes classes, methods, and events to work with file I/O. |
| [util](https://nodejs.org/api/util.html) | util module includes utility functions useful for programmers. |

### Loading Core Modules

In order to use Node.js core or NPM modules, you first need to import it using require() function as shown below.

var module = require('module\_name');

As per above syntax, specify the module name in the require() function. The require() function will return an object, function, property or any other JavaScript type, depending on what the specified module returns.

The following example demonstrates how to use Node.js http module to create a web server.

Example: Load and Use Core http Module

var http = require('http');

var server = http.createServer(function(req, res){

//write code here

});

server.listen(5000);

In the above example, require() function returns an object because http module returns its functionality as an object, you can then use its properties and methods using dot notation e.g. http.createServer().

In this way, you can load and use Node.js core modules in your application. We will be using core modules throughout these tutorials.

# Node.js Local Module/ Custom Module

Local modules are modules created locally in your Node.js application. These modules include different functionalities of your application in separate files and folders. You can also package it and distribute it via NPM, so that Node.js community can use it. For example, if you need to connect to MongoDB and fetch data then you can create a module for it, which can be reused in your application.

## Writing Simple Module

Let's write simple logging module which logs the information, warning or error to the console.

In Node.js, module should be placed in a separate JavaScript file. So, create a Log.js file and write the following code in it.

Log.js

var log = {

info: function (info) {

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

},

warning:function (warning) {

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

},

error:function (error) {

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

}

};

module.exports = log

In the above example of logging module, we have created an object with three functions - info(), warning() and error(). At the end, we have assigned this object to **module.exports**. The module.exports in the above example exposes a log object as a module.

The *module.exports* is a special object which is included in every JS file in the Node.js application by default. Use **module.exports** or **exports** to expose a function, object or variable as a module in Node.js.

Now, let's see how to use the above logging module in our application.

## Loading Local Module

To use local modules in your application, you need to load it using require() function in the same way as core module. However, you need to specify the path of JavaScript file of the module.

The following example demonstrates how to use the above logging module contained in Log.js.

app.js

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

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

In the above example, app.js is using log module. First, it loads the logging module using require() function and specified path where logging module is stored. Logging module is contained in Log.js file in the root folder. So, we have specified the path './Log.js' in the require() function. The '.' denotes a root folder.

The require() function returns a log object because logging module exposes an object in Log.js using module.exports. So now you can use logging module as an object and call any of its function using dot notation e.g myLogModule.info() or myLogModule.warning() or myLogModule.error()

Run the above example using command prompt (in Windows) as shown below.

C:\> node app.js   
Info: Node.js started

Thus, you can create a local module using module.exports and use it in your application.

# Export Module in Node.js

In the previous section, you learned how to write a local module using module.exports. In this section, you will learn how to expose different types as a module using module.exports.

The **module.exports** or **exports** is a special object which is included in every JS file in the Node.js application by default. *module* is a variable that represents current module and *exports* is an object that will be exposed as a module. So, whatever you assign to *module.exports* or *exports*, will be exposed as a module.

Let's see how to expose different types as a module using module.exports.

## Export Literals

As mentioned above, *exports* is an object. So it exposes whatever you assigned to it as a module. For example, if you assign a string literal then it will expose that string literal as a module.

The following example exposes simple string message as a module in Message.js.

Message.js

module.exports = 'Hello world';

//or

exports = 'Hello world';

Now, import this message module and use it as shown below.

app.js

var msg = require('./Messages.js');

console.log(msg);

Run the above example and see the result as shown below.

C:\> node app.js   
Hello World

Note: You must specify './' as a path of root folder to import a local module. However, you do not need to specify path to import Node.js core module or NPM module in the require() function.

## Export Object

*exports* is an object. So, you can attach properties or methods to it. The following example exposes an object with a string property in Message.js file.

Message.js

exports.SimpleMessage = 'Hello world';

//or

module.exports.SimpleMessage = 'Hello world';

In the above example, we have attached a property "SimpleMessage" to the exports object. Now, import and use this module as shown below.

app.js

var msg = require('./Messages.js');

console.log(msg.SimpleMessage);

In the above example, require() function will return an object { SimpleMessage : 'Hello World'}and assign it to the msg variable. So, now you can use msg.SimpleMessage.

Run the above example by writing node app.js in the command prompt and see the output as shown below.

C:\> node app.js   
Hello World

The same way as above, you can expose an object with function. The following example exposes an object with log function as a module.

Log.js

module.exports.log = function (msg) {

console.log(msg);

};

The above module will expose an object- { log : function(msg){ console.log(msg); } } . Use the above module as shown below.

app.js

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

msg.log('Hello World');

Run and see the output in command prompt as shown below.

C:\> node app.js   
Hello World

You can also attach an object to module.exports as shown below.

data.js

module.exports = {

firstName: 'James',

lastName: 'Bond'

}

app.js

var person = require('./data.js');

console.log(person.firstName + ' ' + person.lastName);

Run the above example and see the result as shown below.

C:\> node app.js   
James Bond

## Export Function

You can attach an anonymous function to exports object as shown below.

Log.js

module.exports = function (msg) {

console.log(msg);

};

Now, you can use the above module as below.

app.js

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

msg('Hello World');

The msg variable becomes function expression in the above example. So, you can invoke the function using parenthesis (). Run the above example and see the output as shown below.

C:\> node app.js   
Hello World

## Export function as a class

In the JavaScript, a function can be treated like a class. The following example exposes a function which can be used like a class.

Person.js

module.exports = function (firstName, lastName) {

this.firstName = firstName;

this.lastName = lastName;

this.fullName = function () {

return this.firstName + ' ' + this.lastName;

}

}

The above module can be used as shown below.

app.js

var person = require('./Person.js');

var person1 = new person('James', 'Bond');

console.log(person1.fullName());

As you can see, we have created a person object using new keyword. Run the above example as below.

C:\> node app.js   
James Bond

In this way, you can export and import a local module created in a separate file under root folder.

Node.js also allows you to create modules in sub folders. Let's see how to load module from sub folders.

## Load Module from Separate Folder

Use the full path of a module file where you have exported it using module.exports. For example, if log module in the log.js is stored under "utility" folder under the root folder of your application then import it as shown below.

app.js

var log = require('./utility/log.js');

In the above example, **.** is for root folder and then specify exact path of your module file. Node.js also allows us to specify the path to the folder without specifying file name. For example, you can specify only utility folder without specifing log.js as shown below.

app.js

var log = require('./utility');

In the above example, Node will search for a package definition file called package.json inside utility folder. This is because Node assumes that this folder is a package and will try to look for a package definition. The package.json file should be in a module directory. The package.json under utility folder specifies the file name using "main" key as below.

./utility/package.json

{

"name" : "log",

"main" : "./log.js"

}

# Node.js Web Server

In this section, we will learn how to create a simple Node.js web server and handle HTTP requests.

To access web pages of any web application, you need a [web server](https://en.wikipedia.org/wiki/Web_server). The web server will handle all the http requests for the web application e.g IIS is a web server for ASP.NET web applications and Apache is a web server for PHP or Java web applications.

Node.js provides capabilities to create your own web server which will handle HTTP requests asynchronously. You can use IIS or Apache to run Node.js web application but it is recommended to use Node.js web server.

## Create Node.js Web Server

Node.js makes it easy to create a simple web server that processes incoming requests asynchronously.

The following example is a simple Node.js web server contained in server.js file.

server.js

var http = require('http'); // 1 - Import Node.js core module

var server = http.createServer(function (req, res) { // 2 - creating server

//handle incomming requests here..

});

server.listen(5000); //3 - listen for any incoming requests

console.log('Node.js web server at port 5000 is running..')

In the above example, we import the http module using require() function. The http module is a core module of Node.js, so no need to install it using NPM. The next step is to call createServer() method of http and specify callback function with request and response parameter. Finally, call listen() method of server object which was returned from createServer() method with port number, to start listening to incoming requests on port 5000. You can specify any unused port here.

Run the above web server by writing node server.js command in command prompt or terminal window and it will display message as shown below.

C:\> node server.js   
Node.js web server at port 5000 is running..

This is how you create a Node.js web server using simple steps. Now, let's see how to handle HTTP request and send response in Node.js web server.

## Handle HTTP Request

The http.createServer() method includes [request](https://nodejs.org/api/http.html#http_http_incomingmessage) and [response](https://nodejs.org/api/http.html#http_class_http_serverresponse) parameters which is supplied by Node.js. The request object can be used to get information about the current HTTP request e.g., url, request header, and data. The response object can be used to send a response for a current HTTP request.

The following example demonstrates handling HTTP request and response in Node.js.

server.js

var http = require('http'); // Import Node.js core module

var server = http.createServer(function (req, res) { //create web server

if (req.url == '/') { //check the URL of the current request

// set response header

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

// set response content

res.write('<html><body><p>This is home Page.</p></body></html>');

res.end();

}

else if (req.url == "/student") {

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

res.write('<html><body><p>This is student Page.</p></body></html>');

res.end();

}

else if (req.url == "/admin") {

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

res.write('<html><body><p>This is admin Page.</p></body></html>');

res.end();

}

else

res.end('Invalid Request!');

});

server.listen(5000); //6 - listen for any incoming requests

console.log('Node.js web server at port 5000 is running..')

In the above example, req.url is used to check the url of the current request and based on that it sends the response. To send a response, first it sets the response header using writeHead() method and then writes a string as a response body using write() method. Finally, Node.js web server sends the response using end() method.

Now, run the above web server as shown below.

C:\> node server.js   
Node.js web server at port 5000 is running..

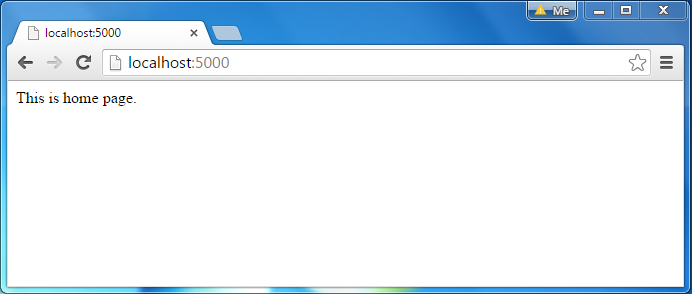
To test it, you can use the command-line program curl, which most Mac and Linux machines have pre-installed.

curl -i http://localhost:5000

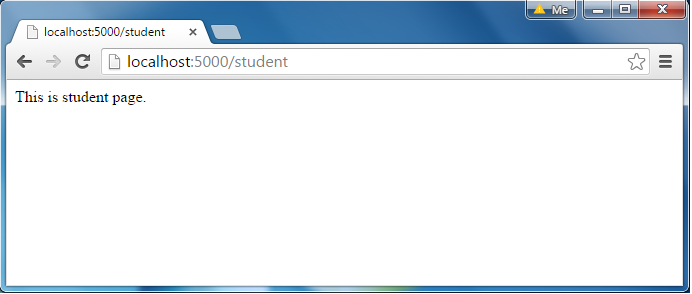
You should see the following response.

HTTP/1.1 200 OK   
Content-Type: text/plain   
Date: Tue, 8 Sep 2015 03:05:08 GMT   
Connection: keep-alive   
This is home page. 

For Windows users, point your browser to *http://localhost:5000* and see the following result.

[](http://www.tutorialsteacher.com/Content/images/nodejs/webserver-response.png)Node.js Web Server Response

The same way, point your browser to *http://localhost:5000/student* and see the following result.

[](http://www.tutorialsteacher.com/Content/images/nodejs/webserver-response2.png)Node.js Web Server Response

It will display "Invalid Request" for all requests other than the above URLs.

## Sending JSON Response

The following example demonstrates how to serve JSON response from the Node.js web server.

server.js

var http = require('http');

var server = http.createServer(function (req, res) {

if (req.url == '/data') { //check the URL of the current request

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

res.write(JSON.stringify({ message: "Hello World"}));

res.end();

}

});

server.listen(5000);

console.log('Node.js web server at port 5000 is running..')

Node.js File System

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

Asynchronous vs synchronous node js

**Synchronous** methods in the **Node**.**js** standard library that use libuv are the most commonly used blocking operations. Native modules may also have blocking methods. All of the I/O methods in the **Node**.**js** standard library provide **asynchronous** versions, which are non-blocking, **and** accept callback functions.

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

Reading File

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

Signature:

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

Parameter Description:

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

The following example demonstrates reading existing TestFile.txt asynchronously.

Example: Reading File

var fs = require('fs');

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

if (err) throw err;

console.log(data);

});

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

The following is a sample TextFile.txt file.

TextFile.txt

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

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

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

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

Example: Reading File Synchronously

var fs = require('fs');

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

console.log(data);

## Writing File

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

Signature:

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

Parameter Description:

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

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

Example: Creating & Writing File (overwrite data to file)

var fs = require('fs');

fs.writeFile('test.txt', 'Hello World!', function (err) {

if (err)

console.log(err);

else

console.log('Write operation complete.');

});

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

Example: Append File Content (previous + new data)

var fs = require('fs');

fs.appendFile('test.txt', 'Hello World!', function (err) {

if (err)

console.log(err);

else

console.log('Append operation complete.');

});

## Open File

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

Signature:

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

Parameter Description:

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

### Flags

The following table lists all the flags which can be used in read/write operation.

| Flag | Description |
| --- | --- |
| R | Open file for reading. An exception occurs if the file does not exist. |
| r+ | Open file for reading and writing. An exception occurs if the file does not exist. |
| Rs | Open file for reading in synchronous mode. |
| rs+ | Open file for reading and writing, telling the OS to open it synchronously. See notes for 'rs' about using this with caution. |
| w | Open file for writing. The file is created (if it does not exist) or truncated (if it exists). |
| wx | Like 'w' but fails if path exists. |
| w+ | Open file for reading and writing. The file is created (if it does not exist) or truncated (if it exists). |
| wx+ | Like 'w+' but fails if path exists. |
| a | Open file for appending. The file is created if it does not exist. |
| ax | Like 'a' but fails if path exists. |
| a+ | Open file for reading and appending. The file is created if it does not exist. |
| ax+ | Like 'a+' but fails if path exists. |

The following example opens an existing file and reads its content.

Example:File open and read

var fs = require('fs');

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

if (err) {

return console.error(err);

}

var buffr = new Buffer(1024);

fs.read(fd, buffr, 0, buffr.length, 0, function (err, bytes) {

if (err) throw err;

// Print only read bytes to avoid junk.

if (bytes > 0) {

console.log(buffr.slice(0, bytes).toString());

}

// Close the opened file.

fs.close(fd, function (err) {

if (err) throw err;

});

});

});

## Delete File

Use fs.unlink() method to delete an existing file.

Signature:

fs.unlink(path, callback);

The following example deletes an existing file.

Example:File Open and Read

var fs = require('fs');

fs.unlink('test.txt', function () {

console.log('write operation complete.');

});

## Important method of fs module

| Method | Description |
| --- | --- |
| fs.readFile(fileName [,options], callback) | Reads existing file. |
| fs.writeFile(filename, data[, options], callback) | Writes to the file. If file exists then overwrite the content otherwise creates new file. |
| fs.open(path, flags[, mode], callback) | Opens file for reading or writing. |
| fs.rename(oldPath, newPath, callback) | Renames an existing file. |
| fs.chown(path, uid, gid, callback) | Asynchronous chown. |
| fs.stat(path, callback) | Returns fs.stat object which includes important file statistics. |
| fs.link(srcpath, dstpath, callback) | Links file asynchronously. |
| fs.symlink(destination, path[, type], callback) | Symlink asynchronously. |
| fs.rmdir(path, callback) | Renames an existing directory. |
| fs.mkdir(path[, mode], callback) | Creates a new directory. |
| fs.readdir(path, callback) | Reads the content of the specified directory. |
| fs.utimes(path, atime, mtime, callback) | Changes the timestamp of the file. |
| fs.exists(path, callback) | Determines whether the specified file exists or not. |
| fs.access(path[, mode], callback) | Tests a user's permissions for the specified file. |
| fs.appendFile(file, data[, options], callback) | Appends new content to the existing file. |

# Buffers

Pure JavaScript is Unicode friendly, but it is not so for binary data. While dealing with TCP streams or the file system, it's necessary to handle octet streams. 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);

### ABCD

65 66 67 68

65 = 1000001

66 =1000010

A =xcc001

X = 674300312v

### 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".

## Writing to 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);

When the above program is executed, it produces the following result −

Octets written : 20

## 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

When the above program is executed, it produces the following result −

abcdefghijklmnopqrstuvwxyz

abcde

abcde

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);

When the above program is executed, it produces the following result −

[ 83, 105, 109, 112, 108, 121, 32, 69, 97, 115, 121, 32, 76, 101, 97, 114, 110, 105, 110,

103 ]

## 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('TutorialsPoint ');

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

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

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

When the above program is executed, it produces the following result −

buffer3 content: TutorialsPoint Simply Easy Learning

## 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);

}

When the above program is executed, it produces the following result −

ABC comes before ABCD

## 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());

When the above program is executed, it produces the following result −

buffer2 content: ABC

## 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('TutorialsPoint');

//slicing a buffer

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

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

When the above program is executed, it produces the following result −

buffer2 content: Tutorials

## 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('TutorialsPoint');

//length of the buffer

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

When the above program is executed, it produces following result −

buffer length: 14

## Methods Reference

Following is a reference of Buffers module available in Node.js. For more detail, you can refer to the official documentation.

## Class Methods

|  |  |
| --- | --- |
| **S.No.** | **Method & Description** |
| 1 | **Buffer.isEncoding(encoding)**  Returns true if the encoding is a valid encoding argument, false otherwise. |
| 2 | **Buffer.isBuffer(obj)**  Tests if obj is a Buffer. |
| 3 | **Buffer.byteLength(string[, encoding])**  Gives the actual byte length of a string. encoding defaults to 'utf8'. It is not the same as String.prototype.length, since String.prototype.length returns the number of characters in a string. |
| 4 | **Buffer.concat(list[, totalLength])**  Returns a buffer which is the result of concatenating all the buffers in the list together. |
| 5 | **Buffer.compare(buf1, buf2)**  The same as buf1.compare(buf2). Useful for sorting an array of buffers. |

## What are 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.

This tutorial provides a basic understanding of the commonly used operations on Streams.

## Reading from a Stream

Create a text file named input.txt having the following content −

Tutorials Point is giving self learning content

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

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");

Now run the main.js to see the result −

$ node main.js

Verify the Output.

Program Ended

Tutorials Point is giving self learning content

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

## Writing to a Stream

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

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");

Now run the main.js to see the result −

$ node main.js

Verify the Output.

Program Ended

Write completed.

Now open output.txt created in your current directory; it should contain the following −

Simply Easy Learning

## 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.

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

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");

Now run the main.js to see the result −

$ node main.js

Verify the Output.

Program Ended

Open output.txt created in your current directory; it should contain the following −

Tutorials Point is giving self learning content

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

## 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.

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

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.");

Now run the main.js to see the result −

$ node main.js

Verify the Output.

File Compressed.

You will find that input.txt has been compressed and it created a file input.txt.gz in the current directory. Now let's try to decompress the same file using the following code −

var fs = require("fs");

var zlib = require('zlib');

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

fs.createReadStream('input.txt.gz')

.pipe(zlib.createGunzip())

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

console.log("File Decompressed.");

Now run the main.js to see the result −

$ node main.js

Verify the Output.

File Decompressed.