Table des matières

[**Node.js Tutorial** 2](#_Toc440633592)

[I. Introduction 2](#_Toc440633593)

[1. Prerequisites 2](#_Toc440633594)

[2. What is Node.js? 2](#_Toc440633595)

[3. Features of Node.js 3](#_Toc440633596)

[4. Who Uses Node.js? 3](#_Toc440633597)

[5. Concepts 3](#_Toc440633598)

[6. Where to Use Node.js? 4](#_Toc440633599)

[7. Where Not to Use Node.js? 4](#_Toc440633600)

[II. Local Environment Setup 5](#_Toc440633601)

[1. Text Editor 5](#_Toc440633602)

[2. The Node.js Runtime 5](#_Toc440633603)

[a) Download Node.js archive 5](#_Toc440633604)

[b) Installation on UNIX/Linux/Mac OS X, and SunOS 6](#_Toc440633605)

[c) Installation on Windows 6](#_Toc440633606)

[d) Verify installation: Executing a File 7](#_Toc440633607)

[III. Node.js - First Application 7](#_Toc440633608)

[1. Creating Node.js Application 8](#_Toc440633609)

[IV. Node.js - REPL Terminal 10](#_Toc440633614)

[V. Node.js - npm 13](#_Toc440633619)

[1. Installing Modules using npm 14](#_Toc440633620)

[2. Global vs Local installation 14](#_Toc440633621)

[3. Uninstalling a module 16](#_Toc440633622)

[4. Updating a module 16](#_Toc440633623)

[5. Search a module 16](#_Toc440633624)

[6. Create a module 16](#_Toc440633625)

[VI. Node.js - Callbacks Concept 17](#_Toc440633626)

[1. What is Callback? 17](#_Toc440633627)

[2. Blocking Code Example 18](#_Toc440633628)

[3. Non-Blocking Code Example 18](#_Toc440633629)

[VII. Node.js - Event Loop 19](#_Toc440633630)

[1. Event Driven Programming 20](#_Toc440633631)

[2. Example 21](#_Toc440633632)

[3. How Node Applications Work? 22](#_Toc440633633)

[VIII. Node.js - Event Emitter 23](#_Toc440633634)

[1. EventEmitter Class 23](#_Toc440633635)

[2. Methods 24](#_Toc440633636)

[3. Class Methods 25](#_Toc440633637)

[4. Events 25](#_Toc440633638)

[5. Example 26](#_Toc440633639)

[IX. Node.js - Buffers 28](#_Toc440633640)

[1. Creating Buffers 28](#_Toc440633641)

[2. Writing to Buffers 29](#_Toc440633645)

[3. Reading from Buffers 29](#_Toc440633650)

[4. Convert Buffer to JSON 31](#_Toc440633655)

[5. Concatenate Buffers 31](#_Toc440633659)

[6. Compare Buffers 32](#_Toc440633664)

[7. Copy Buffer 33](#_Toc440633669)

[8. Slice Buffer 34](#_Toc440633674)

[9. Buffer Length 35](#_Toc440633679)

[10. Methods Reference 35](#_Toc440633683)

[X. Node.js - Streams 36](#_Toc440633684)

[1. What are Streams? 36](#_Toc440633685)

[2. Reading from stream 37](#_Toc440633686)

[3. Writing to stream 38](#_Toc440633687)

[4. Piping streams 39](#_Toc440633688)

[5. Chaining streams 40](#_Toc440633689)

[XI. Node.js - File System 42](#_Toc440633690)

[1. Synchronous vs Asynchronous 42](#_Toc440633691)

[2. Open a File 43](#_Toc440633693)

[3. Flags 44](#_Toc440633696)

[1. Get File information 45](#_Toc440633698)

[4. Writing File 48](#_Toc440633702)

[5. Reading File 49](#_Toc440633706)

[6. Closing File 51](#_Toc440633710)

[7. Truncate File 53](#_Toc440633714)

[8. Delete File 55](#_Toc440633718)

[9. Create Directory 56](#_Toc440633722)

[10. Read Directory 57](#_Toc440633726)

[11. Remove Directory 58](#_Toc440633730)

[XII. Node.js - Web Module 60](#_Toc440633734)

[1. What is Web Server? 60](#_Toc440633735)

[2. Web Application Architecture 60](#_Toc440633736)

[3. Creating Web Server using Node 61](#_Toc440633737)

[4. Creating Web client using Node 64](#_Toc440633739)

[I. Node.js - Express Framework 66](#_Toc440633740)

[1. Express Overview 66](#_Toc440633741)

[2. Installing Express 66](#_Toc440633742)

[3. Hello world Example 67](#_Toc440633743)

[4. Request & Response 68](#_Toc440633744)

[5. Basic Routing 69](#_Toc440633745)

[6. Serving Static Files 71](#_Toc440633746)

[7. GET Method 73](#_Toc440633747)

[8. POST Method 75](#_Toc440633748)

[9. File Upload 77](#_Toc440633749)

[10. Cookies Management 80](#_Toc440633750)

[II. Node.js - RESTful API 80](#_Toc440633751)

[1. What is REST architecture? 80](#_Toc440633752)

[1. RESTful Web Services 81](#_Toc440633754)

[2. Creating RESTful for A Library 81](#_Toc440633755)

[3. List Users 83](#_Toc440633756)

[4. Add User 84](#_Toc440633757)

[5. Show Detail 87](#_Toc440633758)

[6. Delete User 88](#_Toc440633759)

[III. Node.js - Scaling Application 89](#_Toc440633760)

[1. The exec() method 90](#_Toc440633761)

[2. The spawn() method 92](#_Toc440633764)

[3. The fork method 95](#_Toc440633767)

**Node.js Tutorial**

# Introduction

Haut du formulaire

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

# Prerequisites

Before proceeding with this tutorial, you should have a basic understanding of JavaScript. As we are going to develop web-based applications using Node.js, it will be good if you have some understanding of other web technologies such as HTML, CSS, AJAX, etc.

# What is 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](http://nodejs.org/) is as follows −

Node.js is a platform built on [Chrome's JavaScript runtime](http://code.google.com/p/v8/) 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

# 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](https://raw.githubusercontent.com/joyent/node/v0.12.0/LICENSE).

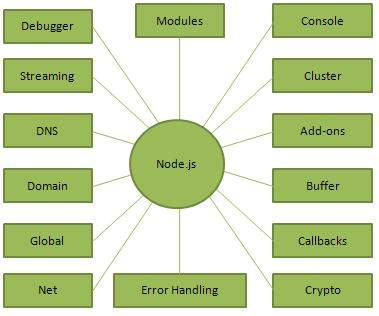
# Who Uses Node.js?

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](https://github.com/joyent/node/wiki/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

# Where Not to Use Node.js?

It is not advisable to use Node.js for CPU intensive applications.

## Local Environment Setup

If you are still willing to set up your environment for Node.js, you need the following two softwares available on your computer, (a) Text Editor and (b) The Node.js binary installables.

## Text Editor

This will be used to type your program. Examples of few editors include Windows Notepad, OS Edit command, Brief, Epsilon, EMACS, and vim or vi.

Name and version of text editor can vary on different operating systems. For example, Notepad will be used on Windows, and vim or vi can be used on windows as well as Linux or UNIX.

The files you create with your editor are called source files and contain program source code. The source files for Node.js programs are typically named with the extension "**.js**".

Before starting your programming, make sure you have one text editor in place and you have enough experience to write a computer program, save it in a file, and finally execute it.

## The Node.js Runtime

The source code written in source file is simply javascript. The Node.js interprter will be used to interpret and execute your javascript code.

Node.js distribution comes as a binary installable for SunOS , Linux, Mac OS X, and Windows operating systems with the 32-bit (386) and 64-bit (amd64) x86 processor architectures.

Following section guides you on how to install Node.js binary distribution on various OS.

## Download Node.js archive

Download latest version of Node.js installable archive file from [Node.js Downloads](http://nodejs.org/download/). At the time of writing this tutorial, following are the versions available on different OS.

|  |  |
| --- | --- |
| **OS** | **Archive name** |
| Windows | node-v0.12.0-x64.msi |
| Linux | node-v0.12.0-linux-x86.tar.gz |
| Mac | node-v0.12.0-darwin-x86.tar.gz |
| SunOS | node-v0.12.0-sunos-x86.tar.gz |

## Installation on UNIX/Linux/Mac OS X, and SunOS

Based on your OS architecture, download and extract the archive node-v0.12.0-**osname**.tar.gz into /tmp, and then finally move extracted files into /usr/local/nodejs directory. For example:

$ cd /tmp

$ wget http://nodejs.org/dist/v0.12.0/node-v0.12.0-linux-x64.tar.gz

$ tar xvfz node-v0.12.0-linux-x64.tar.gz

$ mkdir -p /usr/local/nodejs

$ mv node-v0.12.0-linux-x64/\* /usr/local/nodejs

Add /usr/local/nodejs/bin to the PATH environment variable.

|  |  |
| --- | --- |
| **OS** | **Output** |
| Linux | export PATH=$PATH:/usr/local/nodejs/bin |
| Mac | export PATH=$PATH:/usr/local/nodejs/bin |
| FreeBSD | export PATH=$PATH:/usr/local/nodejs/bin |

## Installation on Windows

Use the MSI file and follow the prompts to install the Node.js. By default, the installer uses the Node.js distribution in C:\Program Files\nodejs. The installer should set the C:\Program Files\nodejs\bin directory in window's PATH environment variable. Restart any open command prompts for the change to take effect.

## 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!

**We will get throught the instruction on how to setup the raspberry pi later**

# Node.js - First Application

Before creating actual "Hello, World!" application using Node.js, let us see the parts of a Node.js application. A Node.js application consists of following three important parts −

* **Import required modules** − We use **require** directive to load a Node.js module.
* **Create server** − A server which will listen to client's request similar to Apache HTTP Server.
* **Read request and return response** − server created in earlier step will read HTTP request made by client which can be a browser or console and return the response.

## Creating Node.js Application

### Step 1 - Import required module

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

var http = require("http");

### Step 2: Create Server

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

http.createServer(function (request, response) {

// Send the HTTP header

// HTTP Status: 200 : OK

// Content Type: text/plain

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

// Send the response body as "Hello World"

response.end('Hello World\n');

}).listen(8081);

// Console will print the message

console.log('Server running at http://127.0.0.1:8081/');

Above code is enough to create an HTTP server which listens ie. wait for a request over 8081 port on local machine.

### Step 3: Testing Request & Response

Let's put step 1 and 2 together in a file called **main.js** and start our HTTP server as shown below −

var http = require("http");

http.createServer(function (request, response) {

// Send the HTTP header

// HTTP Status: 200 : OK

// Content Type: text/plain

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

// Send the response body as "Hello World"

response.end('Hello World\n');

}).listen(8081);

// Console will print the message

console.log('Server running at http://127.0.0.1:8081/');

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

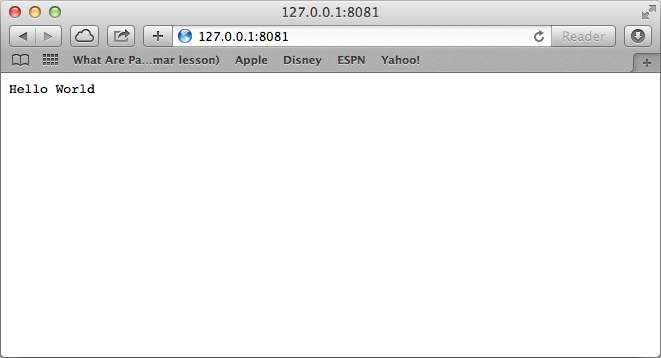
$ node main.js

Verify the Output. Server has started

Server running at http://127.0.0.1:8081/

## Make a request to Node.js server

Open http://127.0.0.1:8081/ in any browser and see the below result.



Congratulations, you have your first HTTP server up and running which is responding all the HTTP requests at port 8081.

# Node.js - REPL Terminal

REPL stands for **Read Eval Print Loop** and it represents a computer environment like a window console or Unix/Linux shell where a command is entered and system responds with an output in interactive mode. Node.js or Node comes bundled with a REPL environment. It performs the following desired tasks.

* **Read** - Reads user's input, parse 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 user press **ctrl-c** twice.

REPL feature of Node is very useful in experimenting with Node.js codes and to debug JavaScript codes.

1. **Starting REPL**

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

$ node

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

$ node

>

### Simple Expression

Let's try simple mathematics at Node.js REPL command prompt:

$ node

> 1 + 3

4

> 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 value is stored in the variable and printed. Whereas if var keyword is used then value is stored but not printed. You can print variables usind 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 enters after opening bracket. Node automatically checks the continuity of expressions.

### Underscore Variable

You can use undercore **\_** 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

>

1. **REPL Commands**

* **ctrl + c** - terminate the current command.
* **ctrl + c twice** - terminate the Node REPL.
* **ctrl + d** - terminate the Node REPL.
* **Up/Down Keys** - see command history and modify previous commands.
* **tab Keys** - list of current commands.
* **.help** - list of all commands.
* **.break** - exit from multiline expression.
* **.clear** - exit from multiline expression
* **.save *filename*** - save current Node REPL session to a file.
* **.load *filename*** - load file content in current Node REPL session.

1. **Stopping REPL**

As mentioned above you will need to use **ctrl + c twice** command to come out of Node.js REPL.

$ node

>

(^C again to quit)

>

# Node.js - npm

Node Package Manager (npm) provides following two main functionalities:

* Online repositories for node.js packages/modules which are searchable on [search.nodejs.org](http://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 installables after v0.6.3 version. To verify the same, open console and type following command and see the result:

$ npm --version

2.7.1

If you are running old version of npm then its damn easy to update it to the latest version. Just use the following command from root:

$ sudo npm install npm -g

/usr/bin/npm -> /usr/lib/node\_modules/npm/bin/npm-cli.js

npm@2.7.1 /usr/lib/node\_modules/npm

## Installing Modules using npm

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

$ npm install <Module Name>

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

$ npm install express

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

var express = require('express');

## Global vs Local installation

By default, npm installs any dependency in the local mode. Here local mode refers to the package installation in node\_modules directory lying in the folder where Node application is present. Locally deployed packages are accessible via require() method. For example when we installed express module, it created node\_modules directory in the current directory where it installed express module.

$ ls -l

total 0

drwxr-xr-x 3 root root 20 Mar 17 02:23 node\_modules

Alternatively you can use **npm ls** command to list down all the locally installed modules.

Globally installed packages/dependencies are stored in system directory. Such dependencies can be used in CLI (Command Line Interface) function of any node.js but can not be imported using require() in Node application directly. Now Let's try installing express module using global installation.

$ npm install express -g

This will produce similar result but module will be installed globally. Here first line tells about the module version and its location where it is getting installed.

**express@4.12.2 /usr/lib/node\_modules/express**

├── merge-descriptors@1.0.0

├── utils-merge@1.0.0

├── cookie-signature@1.0.6

├── methods@1.1.1

├── fresh@0.2.4

├── cookie@0.1.2

├── escape-html@1.0.1

├── range-parser@1.0.2

├── content-type@1.0.1

├── finalhandler@0.3.3

├── vary@1.0.0

├── parseurl@1.3.0

├── content-disposition@0.5.0

├── path-to-regexp@0.1.3

├── depd@1.0.0

├── qs@2.3.3

├── on-finished@2.2.0 (ee-first@1.1.0)

├── etag@1.5.1 (crc@3.2.1)

├── debug@2.1.3 (ms@0.7.0)

├── proxy-addr@1.0.7 (forwarded@0.1.0, ipaddr.js@0.1.9)

├── send@0.12.1 (destroy@1.0.3, ms@0.7.0, mime@1.3.4)

├── serve-static@1.9.2 (send@0.12.2)

├── accepts@1.2.5 (negotiator@0.5.1, mime-types@2.0.10)

└── type-is@1.6.1 (media-typer@0.3.0, mime-types@2.0.10)

You can use following command to check all the modules installed globally:

$ npm ls -g

## Uninstalling a module

Use following command to uninstall a Node.js module.

$ npm uninstall express

Once npm uninstall the package, you can verify by looking at the content of /node\_modules/ directory or type the following command:

$ npm ls

## Updating a module

Update package.json and change the version of the dependency which to be updated and run the following command.

$ npm update express

## Search a module

Search package name using npm.

$ npm search express

## Create a module

Creation of module requires package.json to be generated. Let's generate package.json using npm, which will generate basic skeleton of the package.json.

$ npm init

This utility will walk you through creating a package.json file.

It only covers the most common items, and tries to guess sane defaults.

See 'npm help json' for definitive documentation on these fields

and exactly what they do.

Use 'npm install <pkg> --save' afterwards to install a package and

save it as a dependency in the package.json file.

Press ^C at any time to quit.

name: (webmaster)

You will need to provide all the required information about your module. YOu can take help from the above mentioned package.json file to understand the meanings of various information demanded. Once package.json is generated. Use the following command to register yourself with npm repository site using a valid email address.

$ npm adduser

Username: mcmohd

Password:

Email: (this IS public) mcmohd@gmail.com

Now its time to publish your module:

$ npm publish

If everything is fine with your module, then it will be published in the reporsitory and will be accessible to install using npm like any other other Node.js module.

# Node.js - Callbacks Concept

## What is Callback?

Callback is an asynchronous equivalent for a function. A callback function is called at the completion of a given task. Node makes heavy use of callbacks. All APIs of Node are written is such a way that they supports callbacks.

For example, a function to read a file may start reading file and return the control to execution environment immidiately so that next instruction can be executed. Once file I/O is complete, it will call the callback function while passing the callback function, the content of the file as parameter. So there is no blocking or wait for File I/O. This makes Node.js highly scalable, as it can process high number of request without waiting for any function to return result.

## Blocking Code Example

Create a text file named input.txt having 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 which has the following code:

var fs = require("fs");

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

console.log(data.toString());

console.log("Program Ended");

Now run the main.js to see the result:

$ node main.js

Verify the Output

Tutorials Point is giving self learning content

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

Program Ended

## Non-Blocking Code Example

Create a text file named input.txt having following content

Tutorials Point is giving self learning content

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

Update main.js file to have following code:

var fs = require("fs");

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

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

console.log(data.toString());

});

console.log("Program Ended");

Now 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!!!!!

These two examples explain the concept of blocking and non-blocking calls. First example shows that program blocks until it reads the file and then only it proceeds to end the program where as in second example, program does not wait for file reading but it just proceeded to print "Program Ended" and same time program without blocking continues reading the file.

Thus, a blocking program executes very much in sequence and from programming point of view its easier to implement the logic but non-blocking programs does not execute in sequence, so in case a program needs to use any data to be processed, it should be kept with-in the same block to make it sequential execution.

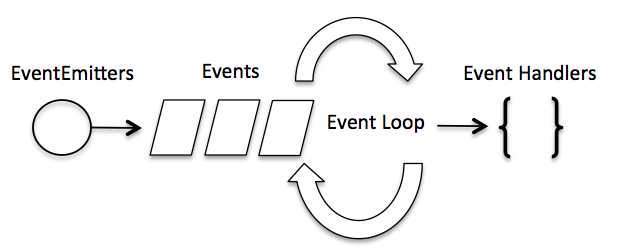
# Node.js - Event Loop

Node js is a single threaded application but it support concurrency via concept of event and callbacks. As every API of Node js are asynchronous and being a single thread, it uses **async** function calls to maintain the concurrency. Node uses observer pattern. Node thread keeps an event loop and whenever any task get completed, it fires the corresponding event which signals the event listener function to get executed.

## 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, delcares functions and then simply waits for 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.



While Events seems similar to what callbacks are. The difference lies in the fact that callback functions are called when an asynchronous function returns its result where as event handling works on the observer pattern. The functions which listens to events acts as Observers. Whenever an event gets fired, its listener function starts executing. Node.js has multiple in-built events available through **events** module and **EventEmitter** class which is used to bind events and event listeners as follows:

// Import events module

var events = require('events');

// Create an eventEmitter object

var eventEmitter = new events.EventEmitter();

Following is the syntax to bind event handler with an event:

// Bind event and even handler as follows

eventEmitter.on('eventName', eventHandler);

We can fire an event programatically as follows:

// Fire an event

eventEmitter.emit('eventName');

## Example

Create a js file named main.js having the following code:

// 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.");

Now let's try to run the above program as check the output:

$ mnode main.js

This will produce following result:

connection succesful.

data received succesfully.

Program Ended.

## How Node Applications Work?

In Node Application, any async function accepts a callback as a last parameter and the callback function accepts error as a first parameter. Let's revisit the previous example again. Create a text file named input.txt having 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 having the following code:

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");

Here fs.readFile() is a async function whose purpose is to read a file. If an error occurs during read of file, then err object will contain the corresponding error else data will contain the contents of the file. readFile passes err and data to callback function after file read operation is complete, which finally prints the content.

Program Ended

Tutorials Point is giving self learning content

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

# Node.js - Event Emitter

Many objects in Node emit events for example a **net.Server** emits an event each time a peer connects to it, a **fs.readStream** emits an event when the file is opened. All objects which emit events are instances of**events.EventEmitter**.

## EventEmitter Class

As we have seen in previous section, EventEmitter class lies in **events** module. It is accessibly via following syntax:

// 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 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

|  |  |
| --- | --- |
| **S.N.** | **method & Description** |
| 1 | **addListener(event, listener)** Adds a listener to the end of the listeners array for the specified event. No checks are made to see if the listener has already been added. Multiple calls passing the same combination of event and listener will result in the listener being added multiple times. Returns emitter, so calls can be chained. |
| 2 | **on(event, listener)** Adds a listener to the end of the listeners array for the specified event. No checks are made to see if the listener has already been added. Multiple calls passing the same combination of event and listener will result in the listener being added multiple times. Returns emitter, so calls can be chained. |
| 3 | **once(event, listener)** Adds a one time listener for the event. This listener is invoked only the next time the event is fired, after which it is removed. Returns emitter, so calls can be chained. |
| 4 | **removeListener(event, listener)** Remove a listener from the listener array for the specified event. Caution: changes array indices in the listener array behind the listener. removeListener will remove, at most, one instance of a listener from the listener array. If any single listener has been added multiple times to the listener array for the specified event, then removeListener must be called multiple times to remove each instance. Returns emitter, so calls can be chained. |
| 5 | **removeAllListeners([event])** Removes all listeners, or those of the specified event. It's not a good idea to remove listeners that were added elsewhere in the code, especially when it's on an emitter that you didn't create (e.g. sockets or file streams). Returns emitter, so calls can be chained. |
| 6 | **setMaxListeners(n)** By default EventEmitters will print a warning if more than 10 listeners are added for a particular event. This is a useful default which helps finding memory leaks. Obviously not all Emitters should be limited to 10. This function allows that to be increased. Set to zero for unlimited. |
| 7 | **listeners(event)** Returns an array of listeners for the specified event. |
| 8 | **emit(event, [arg1], [arg2], [...])** Execute each of the listeners in order with the supplied arguments. Returns true if event had listeners, false otherwise. |

## Class Methods

|  |  |
| --- | --- |
| **S.N.** | **method & Description** |
| 1 | **listenerCount(emitter, event)** Return the number of listeners for a given event. |

## Events

|  |  |
| --- | --- |
| **S.No.** | **Events & Description** |
| 1 | **newListener**   * **event** - String The event name * **listener** - Function The event handler function   This event is emitted any time a listener is added. When this event is triggered, the listener may not yet have been added to the array of listeners for the event. |
| 2 | **removeListener**   * **event** - String The event name * **listener** - Function The event handler function   This event is emitted any time someone removes a listener. When this event is triggered, the listener may not yet have been removed from the array of listeners for the event. |

## Example

Create a js file named main.js with the following Node.js code:

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

Pure JavaScript is Unicode friendly but not nice to binary data. When 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 and can be accessed in application without importing 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");

Though "utf8" is the default encoding but you can use either of the 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 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 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 above program is executed, it produces 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 above program is executed, it produces 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 above program is executed, it produces 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 above program is executed, it produces 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 this 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 above program is executed, it produces 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 above program is executed, it produces 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, 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 above program is executed, it produces 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 a size of buffer in bytes.

### Example

var buffer = new Buffer('TutorialsPoint');

//length of the buffer

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

When 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 a further detail you can refer to official documentation.

|  |  |
| --- | --- |
| **SN** | **Method & Description** |
| 1 | **new Buffer(size)**  Allocates a new buffer of size octets. Note, size must be no more than kMaxLength. Otherwise, a RangeError will be thrown here. |
| 2 | **new Buffer(buffer)**  Copies the passed buffer data onto a new Buffer instance. |
| 3 | **new Buffer(str[, encoding])** Allocates a new buffer containing the given str. encoding defaults to 'utf8'. |
| 4 | **buf.length** Returns the size of the buffer in bytes. Note that this is not necessarily the size of the contents. length refers to the amount of memory allocated for the buffer object. It does not change when the contents of the buffer are changed. |
| 5 | **buf.write(string[, offset][, length][, encoding])** Writes string to the buffer at offset using the given encoding. offset defaults to 0, encoding defaults to 'utf8'. length is the number of bytes to write. Returns number of octets written. |
| 6 | **buf.writeUIntLE(value, offset, byteLength[, noAssert])** Writes value to the buffer at the specified offset and byteLength. Supports up to 48 bits of accuracy. Set noAssert to true to skip validation of value and offset. Defaults to false. |

# Node.js - Streams

## What are Streams?

Streams are objects that let you read data from a source or write data to a destination in continous 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 data has been flushed to underlying system

This tutorial will give you understanding on commonly used operations on Streams.

## Reading from stream

Create a text file named input.txt having 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 which has 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 stream

Create a js file named main.js which has 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 and verify the following content available in output.txt file.

Simply Easy Learning

## Piping streams

Piping is a mechanism where we provide output of one stream as the input to another stream. It is normally used to get data from one stream and to pass 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 which has 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 and verify the following content available in output.txt file.

Tutorials Point is giving self learning content

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

## Chaining streams

Chanining is a mechanism to connect 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 the piping and chaining to first compress a file and then decompress the same.

Create a js file named main.js which has 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 file 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.

# Node.js - File System

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

var fs = require("fs")

## Synchronous vs Asynchronous

Every method in fs module have synchronous as well as asynchronous form. Asynchronous methods takes a last parameter as completion function callback and first parameter of the callback function is error. It is preferred to use asynchronous method instead of synchronous method as former never block the program execution where as the second one does.

### Example

Create a text file named **input.txt** having following content

Tutorials Point is giving self learning content

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

Let us create a js file named **main.js** having the following code.

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");

Now run the main.js to see the result:

$ node main.js

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!!!!!

Following section will give good examples on major File I/O methods.

## Open a File

### Syntax

Following is the syntax of the method to open a file in asynchronous mode:

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

### Parameters

Here is the description of the parameters used:

* **path** - This is string having file name including path.
* **flags** - Flag tells the behavior of the file to be opened. All possible values have been mentioned below.
* **mode** - This 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:

|  |  |
| --- | --- |
| **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. |

### Example

Let us create a js file named **main.js** having the following code to open a file input.txt for reading and writing.

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

Following is the syntax of the method to get the information about a file:

fs.stat(path, callback)

### Parameters

Here is the description of the parameters used:

* **path** - This is 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.

Apart from the important attributes which are printed below in the example, there are number of useful methods available in **fs.Stats** class which can be used to check file type. These methods are given in the following table.

|  |  |
| --- | --- |
| **Method** | **Description** |
| stats.isFile() | Returns true if file type of a simple file. |
| stats.isDirectory() | Returns true if file type of a directory. |
| stats.isBlockDevice() | Returns true if file type of a block device. |
| stats.isCharacterDevice() | Returns true if file type of a character device. |
| stats.isSymbolicLink() | Returns true if file type of a symbolic link. |
| stats.isFIFO() | Returns true if file type of a FIFO. |
| stats.isSocket() | Returns true if file type of asocket. |

### Example

Let us create a js file named **main.js** having the following code:

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 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 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 string having 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 and used to to return error in case of any writing error.

### Example

Let us create a js file named **main.js** having the following code:

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

Going to write into existing file

Data written successfully!

Let's read newly written data

Asynchronous read: Simply Easy Learning!

## Reading File

### Syntax

Following is the syntax of one of the methods to read from a file:

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

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

### Parameters

Here is the description of the parameters used:

* **fd** - This is the file descriptor returned by file fs.open() method.
* **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

Let us create a js file named **main.js** having the following code:

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());

}

});

});

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 read the file

97 bytes read

Tutorials Point is giving self learning content

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

## Closing File

### Syntax

Following is the syntax of one of the methods to close an opened file:

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 which gets no arguments other than a possible exception are given to the completion callback.

### Example

Let us create a js file named **main.js** having the following code:

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 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 read the file

Tutorials Point is giving self learning content

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

File closed successfully.

## Truncate File

### Syntax

Following is the syntax of the method to truncate an opened file:

fs.ftruncate(fd, len, callback)

### Parameters

Here is the description of the parameters used:

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

### Example

Let us create a js file named **main.js** having the following code:

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 File

### Syntax

Following is the syntax of the method to delete a file:

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 which gets no arguments other than a possible exception are given to the completion callback.

### Example

Let us create a js file named **main.js** having the following code:

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

Verify the Output

Going to delete an existing file

File deleted successfully!

## Create Directory

### Syntax

Following is the syntax of the method to create a directory:

fs.mkdir(path[, mode], callback)

### Parameters

Here is the description of the parameters used:

* **path** - This is the directory name including path.
* **mode** - This is the directory permission to be set. Defaults to 0777.
* **callback** - This is the callback function which gets no arguments other than a possible exception are given to the completion callback.

### Example

Let us create a js file named **main.js** having the following code:

var fs = require("fs");

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

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

if (err) {

return console.error(err);

}

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

});

Now run the main.js to see the result:

$ node main.js

Verify the Output

Going to create directory /tmp/test

Directory created successfully!

## Read Directory

### Syntax

Following is the syntax of the method to read a directory:

fs.readdir(path, callback)

### Parameters

Here is the description of the parameters used:

* **path** - This is the directory name including path.
* **callback** - This is the callback function which gets two arguments (err, files) where files is an array of the names of the files in the directory excluding '.' and '..'.

### Example

Let us create a js file named **main.js** having the following code:

var fs = require("fs");

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

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

if (err) {

return console.error(err);

}

files.forEach( function (file){

console.log( file );

});

});

Now run the main.js to see the result:

$ node main.js

Verify the Output

Going to read directory /tmp

ccmzx99o.out

ccyCSbkF.out

employee.ser

hsperfdata\_apache

test

test.txt

## Remove Directory

### Syntax

Following is the syntax of the method to remove a directory:

fs.rmdir(path, callback)

### Parameters

Here is the description of the parameters used:

* **path** - This is the directory name including path.
* **callback** - This is the callback function which gets no arguments other than a possible exception are given to the completion callback.

### Example

Let us create a js file named **main.js** having the following code:

var fs = require("fs");

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

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

if (err) {

return console.error(err);

}

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

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

if (err) {

return console.error(err);

}

files.forEach( function (file){

console.log( file );

});

});

});

Now run the main.js to see the result:

$ node main.js

Verify the Output

Going to read directory /tmp

ccmzx99o.out

ccyCSbkF.out

employee.ser

hsperfdata\_apache

test.txt

# Node.js - Web Module

## What is Web Server?

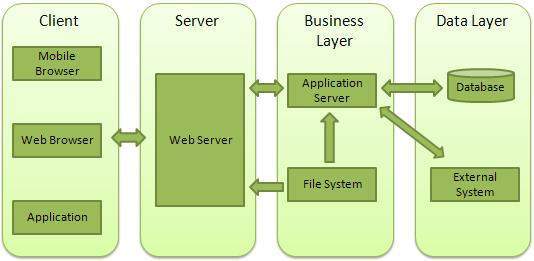
Web Server is a software application which handles HTTP requests sent by the HTTP client, like web browsers, and returns web pages in response to the clients. Web servers usually delivers html documents along with images, style sheets and scripts.

Most of the web server support server side scripts using scripting language or redirect to application server which perform the specific task of getting data from database, perform complex logic etc. and then sends a result to the HTTP client through the Web server.

Apache web server is one of the most commonly used web server. 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 request to the web server.
* **Server** - This layer consists of Web server which can intercepts the request made by clients and pass them the response.
* **Business** - This layer consists of application server which is utilized by web server to do required processing. This layer interacts with data layer via data base or some external programs.
* **Data** - This layer consists of databases or any source of data.

## Creating Web Server using Node

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

Create a js file named server.js:

*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 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:

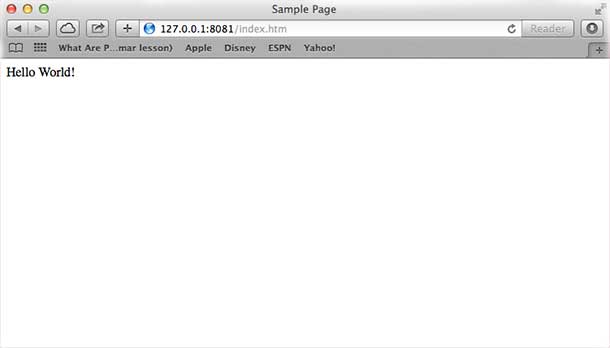
$ node server.js

Verify the Output

Server running at http://127.0.0.1:8081/

## Make a request to Node.js server

Open http://127.0.0.1:8081/index.htm in any browser and see the below result.



Verify the Output at server end.

Server running at http://127.0.0.1:8081/

Request for /index.htm received.

## Creating Web client using Node

A web client can be created using **http** module. Let's check the following example.

Create a js file named client.js:

*File: client.js*

var http = require('http');

// Options to be used by request

var options = {

host: 'localhost',

port: '8081',

path: '/index.htm'

};

// Callback function is used to deal with response

var callback = function(response){

// Continuously update stream with data

var body = '';

response.on('data', function(data) {

body += data;

});

response.on('end', function() {

// Data received completely.

console.log(body);

});

}

// Make a request to the server

var req = http.request(options, callback);

req.end();

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

$ node client.js

Verify the Output.

<html>

<head>

<title>Sample Page</title>

</head>

<body>

Hello World!

</body>

</html>

Verify the Output at server end.

Server running at http://127.0.0.1:8081/

Request for /index.htm received.

# Node.js - 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 a rapid development of Node based Web applications. Following are some of the core features of Express framework:

* Allows to set up middlewares to respond to HTTP Requests.
* Defines a routing table which is used to perform different action based on HTTP Method and URL.
* Allows to dynamically render 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 web application using node terminal.

$ npm install express --save

Above command saves installation locally in **node\_modules** directory and creates a directory express inside node\_modules. There are following important modules which you should install 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 3000 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)

})

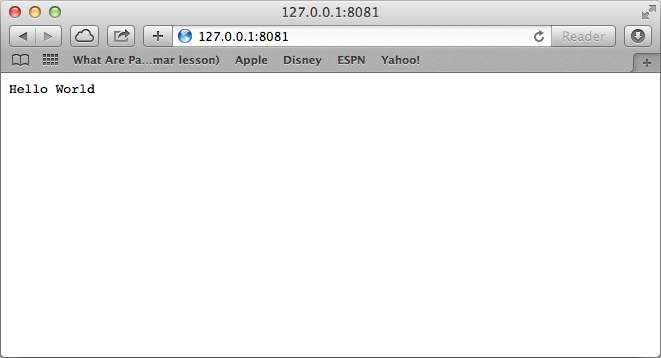
Save the above code in a file named server.js and run it with the following command.

$ node server.js

You will see the following output:

Example app listening at http://0.0.0.0:8081

Open http://127.0.0.1:8081/ in any browser and see the below result.



## Request & Response

Express application makes use of a callback function whose parameters are**request** and **response** objects.

app.get('/', function (req, res) {

// --

})

You can check further detail on both the objects:

* [Request Object](http://www.tutorialspoint.com/nodejs/nodejs_request_object.htm) - The request object represents the HTTP request and has properties for the request query string, parameters, body, HTTP headers, and so on.
* [Response Object](http://www.tutorialspoint.com/nodejs/nodejs_response_object.htm) - The response object represents the HTTP response that an Express app sends when it gets an HTTP request.

You can print **req** and **res** objects which provide lot of information related to HTTP request and response including cookies, sessions, URL etc.

## Basic Routing

We have seen a basic application which serves HTTP request for the homepage. Routing refers to determining how an application responds to a client request to a particular endpoint, which is a URI (or path) and a specific HTTP request method (GET, POST, and so on).

We will extend our Hello World program to add functionality to handle more type HTTP requests.

var express = require('express');

var app = express();

// This responds with "Hello World" on the homepage

app.get('/', function (req, res) {

console.log("Got a GET request for the homepage");

res.send('Hello GET');

})

// This responds a POST request for the homepage

app.post('/', function (req, res) {

console.log("Got a POST request for the homepage");

res.send('Hello POST');

})

// This responds a DELETE request for the /del\_user page.

app.delete('/del\_user', function (req, res) {

console.log("Got a DELETE request for /del\_user");

res.send('Hello DELETE');

})

// This responds a GET request for the /list\_user page.

app.get('/list\_user', function (req, res) {

console.log("Got a GET request for /list\_user");

res.send('Page Listing');

})

// This responds a GET request for abcd, abxcd, ab123cd, and so on

app.get('/ab\*cd', function(req, res) {

console.log("Got a GET request for /ab\*cd");

res.send('Page Pattern Match');

})

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

You will see the following output:

Example app listening at http://0.0.0.0:8081

Now you can try different requests at http://127.0.0.1:8081 to see the output generated by server.js. Following are few screens showing different responses for different URLs.

Screen showing again http://127.0.0.1:8081/list\_user

Screen showing again http://127.0.0.1:8081/abcd

Screen showing again http://127.0.0.1:8081/abcdefg

## Serving Static Files

Express provides a built-in middleware **express.static** to serve static files, such as images, CSS, JavaScript etc.

You simply needs to pass the name of the directory where you keep your static assets, to the **express.static** middleware to start serving the files directly. For example, if you keep your images, CSS, and JavaScript files in a directory named public, you can do this:

app.use(express.static('public'));

We will keep few images in **public/images** sub-directory as follows:

node\_modules

server.js

public/

public/images

public/images/logo.png

Let's modify "Hello Word" app to add the functionality to handle static files.

var express = require('express');

var app = express();

app.use(express.static('public'));

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

Now open http://127.0.0.1:8081/images/logo.png in any browser and see the below result.

## GET Method

Here is a simple example which passes two values using HTML FORM GET method. We are going to use **process\_get** router inside server.js to handle this input.

<html>

<body>

<form action="http://127.0.0.1:8081/process\_get" method="GET">

First Name: <input type="text" name="first\_name"> <br>

Last Name: <input type="text" name="last\_name">

<input type="submit" value="Submit">

</form>

</body>

</html>

Let's save above code in index.htm and modify server.js to handle home page request as well as input sent by the HTML form.

var express = require('express');

var app = express();

app.use(express.static('public'));

app.get('/index.htm', function (req, res) {

res.sendFile( \_\_dirname + "/" + "index.htm" );

})

app.get('/process\_get', function (req, res) {

// Prepare output in JSON format

response = {

first\_name:req.query.first\_name,

last\_name:req.query.last\_name

};

console.log(response);

res.end(JSON.stringify(response));

})

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)

})

Now accessing HTML document using *http://127.0.0.1:8081/index.htm* will generate following form:

Haut du formulaire

|  |  |
| --- | --- |
| First Name: |  |
| Last Name: |  |
|  |  |
|  | |

Bas du formulaire

Now you can enter First and Last Name and then click submit button to see the result and it should give result as follows:

{"first\_name":"John","last\_name":"Paul"}

## POST Method

Here is a simple example which passes two values using HTML FORM POST method. We are going to use **process\_get** router inside server.js to handle this input.

<html>

<body>

<form action="http://127.0.0.1:8081/process\_post" method="POST">

First Name: <input type="text" name="first\_name"> <br>

Last Name: <input type="text" name="last\_name">

<input type="submit" value="Submit">

</form>

</body>

</html>

Let's save above code in index.htm and modify server.js to handle home page request as well as input sent by the HTML form.

var express = require('express');

var app = express();

var bodyParser = require('body-parser');

// Create application/x-www-form-urlencoded parser

var urlencodedParser = bodyParser.urlencoded({ extended: false })

app.use(express.static('public'));

app.get('/index.htm', function (req, res) {

res.sendFile( \_\_dirname + "/" + "index.htm" );

})

app.post('/process\_post', urlencodedParser, function (req, res) {

// Prepare output in JSON format

response = {

first\_name:req.body.first\_name,

last\_name:req.body.last\_name

};

console.log(response);

res.end(JSON.stringify(response));

})

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)

})

Now accessing HTML document using *http://127.0.0.1:8081/index.htm* will generate following form:

Haut du formulaire

|  |  |
| --- | --- |
| First Name: |  |
| Last Name: |  |
|  | |

Bas du formulaire



Now you can enter First and Last Name and then click submit button to see the result and it should give result as follows:

{"first\_name":"John","last\_name":"Paul"}

## File Upload

The following HTML code creates a file uploader form. This form is having method attribute set to **POST** and enctype attribute is set to **multipart/form-data**

<html>

<head>

<title>File Uploading Form</title>

</head>

<body>

<h3>File Upload:</h3>

Select a file to upload: <br />

<form action="http://127.0.0.1:8081/file\_upload" method="POST"

enctype="multipart/form-data">

<input type="file" name="file" size="50" />

<br />

<input type="submit" value="Upload File" />

</form>

</body>

</html>

Let's save above code in index.htm and modify server.js to handle home page request as well as file upload.

var express = require('express');

var app = express();

var fs = require("fs");

var bodyParser = require('body-parser');

var multer = require('multer');

app.use(express.static('public'));

app.use(bodyParser.urlencoded({ extended: false }));

app.use(multer({ dest: '/tmp/'}));

app.get('/index.htm', function (req, res) {

res.sendFile( \_\_dirname + "/" + "index.htm" );

})

app.post('/file\_upload', function (req, res) {

console.log(req.files.file.name);

console.log(req.files.file.path);

console.log(req.files.file.type);

var file = \_\_dirname + "/" + req.files.file.name;

fs.readFile( req.files.file.path, function (err, data) {

fs.writeFile(file, data, function (err) {

if( err ){

console.log( err );

}else{

response = {

message:'File uploaded successfully',

filename:req.files.file.name

};

}

console.log( response );

res.end( JSON.stringify( response ) );

});

});

})

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)

})

Now accessing HTML document using *http://127.0.0.1:8081/index.htm* will generate following form:

**File Upload:**

Select a file to upload:

NOTE: This is just dummy form and would not work, but it must work at your server.

## Cookies Management

You can send cookies to Node.js server which can handle the using the following middleware option. Following is a simple example to print all the cookies sent by the client.

var express = require('express')

var cookieParser = require('cookie-parser')

var app = express()

app.use(cookieParser())

app.get('/', function(req, res) {

console.log("Cookies: ", req.cookies)

})

app.listen(8081)

# Node.js - RESTful API

## What is REST architecture?

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 representation 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.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **S. N.** | **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. |

I'm keeping most of the part of all the examples in the form of hard coding assuming you already know how to pass values from front end using Ajax or simple form data and how to process them using express **Request** object.

## List Users

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

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)

})

Now try to access defined API using *http://127.0.0.1:8081/listUsers* on local machine. 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. Following is the detail of the new user:

user = {

"user4" : {

"name" : "mohit",

"password" : "password4",

"profession" : "teacher",

"id": 4

}

}

You can accept the same input in the form of JSON using Ajax call but for teaching point of view, we are making it hard coded here. Following is the**addUser** API to a new user in the database:

var express = require('express');

var app = express();

var fs = require("fs");

var user = {

"user4" : {

"name" : "mohit",

"password" : "password4",

"profession" : "teacher",

"id": 4

}

}

app.get('/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)

})

Now try to access defined API using *http://127.0.0.1:8081/addUsers* on local machine. This should produce following result:

{ 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.

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) {

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)

})

Now let's call above service using *http://127.0.0.1:8081/2* on local machine. 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.

var express = require('express');

var app = express();

var fs = require("fs");

var id = 2;

app.get('/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)

})

Now let's call above service using *http://127.0.0.1:8081/deleteUser* on local machine. This should produce following result:

{ user1:

{ name: 'mahesh',

password: 'password1',

profession: 'teacher',

id: 1 },

user3:

{ name: 'ramesh',

password: 'password3',

profession: 'clerk',

id: 3 }

}

# Node.js - Scaling Application

As 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 following three major ways to create 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** Function gets three arguments **error**, **stdout** and **stderr**which is called with the following output when 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 file 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);

});

}

Now run the master.js to see the result:

$ node master.js

Verify the 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 large amount of data. spawn() starts receiving the response as soon as the process starts executing.

## Example

Create two js file 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.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);

});

}

Now run the master.js to see the result:

$ node master.js

Verify the 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 the 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 object with a built-in communication channel in addition to having all the methods in a normal ChildProcess instance.

## Example

Create two js file 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);

});

}

Now run the master.js to see the result:

$ node master.js

Verify the 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

Bas du formulaire