Node.js - Quick Guide

Previous Page Next Page

Node.js - Introduction

What is Node.js?

Node.js is a web application framework built on Google Chrome's JavaScript Engine(V8 Engine). Its latest version is v0.10.36. Defintion of Node.js as put by its official documentation is as follows:

> Node.js® is a platform built on Chrome's JavaScript runtime for easily building fast, 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 comes with runtime environment on which a Javascript based script can be interpreted and executed (It is analogus to JVM to JAVA byte code). This runtime allows to execute a JavaScript code on any machine outside a browser. Because of this runtime of Node.js, JavaScript is now can be executed on server as well.

Node.js also provides a rich library of various javascript modules which eases the developement of web application using Node.js to great extents.

Node.js = Runtime Environment + JavaScript Library

Features of Node.js

Aynchronous and Event Driven All APIs of Node.js library are aynchronous that is non-blocking. It essentially means a Node.js based server never waits for a API to return data. Server moves to next API after calling it and a notification mechanism of Events of Node.js helps server to

get 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 server to respond in a non-bloking ways and makes server highly scalable as opposed to traditional servers which create limited threads to handle requests. Node.js uses a single threaded program and same program can services much larger number of requests than traditional server 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 .

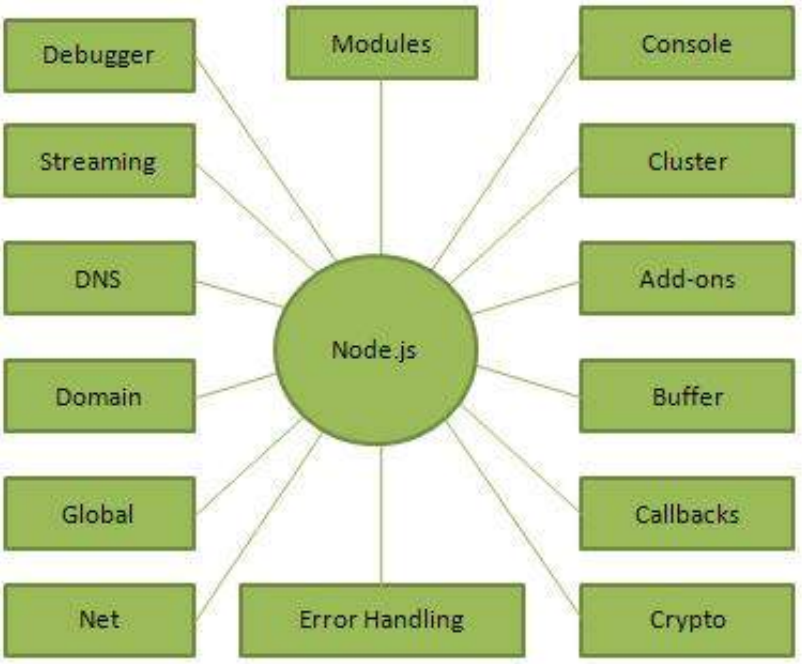
Who Uses Node.js?

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

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 a perfect technology partner.

I/O bound Applications

Data Streaming Applications

Data Intensive Realtime 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.

Node.js - Environment Setup

Try it Option Online

You really do not need to set up your own environment to start learning Node.js. Reason is very simple, we already have set up Node.js environment online, so that you can compile and execute all the available examples online at the same time when you are doing your theory work. This gives you confidence in what you are reading and to check the result with different options. Feel free to modify any example and execute it online.

Try following example using **Try it** option available at the top right corner of the below sample code box:

```
console.log("Hello World!");
```

For most of the examples given in this tutorial, you will find **Try it** option, so just make use of it and enjoy your learning.

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, compile it and finally execute it.

The Node.js Runtime

The source code written in source file is simply javascript. The Node.js interpreter 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 . At the time of writing this tutorial, I downloaded *node-v0.12.0-x64.msi* and copied it into C:\>nodejs folder.

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

Extract the download archive into /usr/local, creating a NodeJs tree in /usr/local/nodejs. For example:

tar -C /usr/local -xzf node-v0.12.0-linux-x86.tar.gz

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

OS	Output
Linux	export PATH=$PATH:/usr/local/nodejs
Mac	export PATH=$PATH:/usr/local/nodejs
FreeBSD	export PATH=$PATH:/usr/local/nodejs

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 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 test.js in **C:\>Nodejs_WorkSpace**.

File: test.js

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

Now run the test.js to see the result:

```
C:\Nodejs_WorkSpace>node test.js
```

Verify the Output

```
Hello, World!
```

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 module: use require directive to load a javascript 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

use require directive to load http module.

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

Step 2: create an HTTP server using http.createServer method. Pass it a function with parameters request and response. Write the sample implementation to always return "Hello World". Pass a port 8081 to listen method.

```
http.createServer(function (request, response) {
   // 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/');
```

Step 3: Create a js file named test.js in **C:\>Nodejs_WorkSpace**.

File: test.js

```
var http = require("http")
http.createServer(function (request, response) {
   response.writeHead(200, {'Content-Type': 'text/plain'});
   response.end('Hello World\n');
}).listen(8081);
console.log('Server running at http://127.0.0.1:8081/');
```

Now run the test.js to see the result:

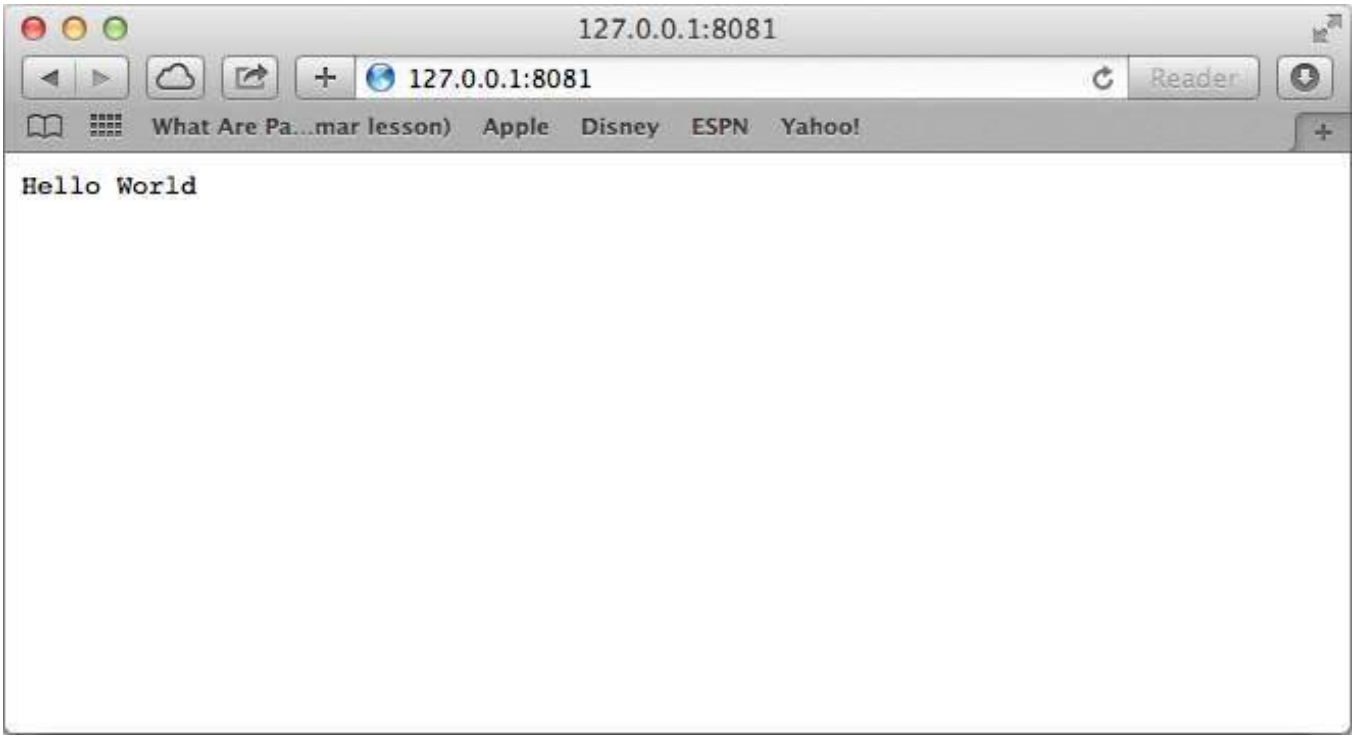
```
C:\Nodejs_WorkSpace>node test.js
```

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.



Node.js - REPL

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. 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.

Features

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

```
C:\Nodejs_WorkSpace> node
```

You will see the REPL Command prompt:

```
C:\Nodejs_WorkSpace> node
>
```

Simple Expression

Let's try simple mathematics at REPL command prompt:

```
C:\Nodejs_WorkSpace>node
> 1 + 3
4
> 1 + ( 2 * 3 ) - 4
3
>
```

Use variables

Use variables to store values and print later. if var keyword is not used then value is stored in the variable and printed. Wheras if var keyword is used then value is stored but not printed. You can use both variables later. Print anything usind console.log()

```
C:\Nodejs_WorkSpace> node
> x = 10
10
> var y = 10
undefined
> x + y
20
> console.log("Hello World")
```

```
Hello Workd
undefined
```

Multiline Expression

Node REPL supports multiline expression similar to JavaScript. See the following do-while loop in action:

```
C:\Nodejs_WorkSpace> 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

Use **_** to get the last result.

```
C:\Nodejs_WorkSpace>node
> var x = 10
undefined
> var y = 20
undefined
> x + y
30
> var sum = _
undefined
> console.log(sum)
30
undefined
>
```

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 - save current Node REPL session to a file.

.load - load file content in current Node REPL session.

```
C:\Nodejs_WorkSpace>node
> var x = 10
undefined
> var y = 20
undefined
> x + y
30
> var sum = _
undefined
> console.log(sum)
30
undefined
> .save test.js
Session saved to:test.js
> .load test.js
> var x = 10
undefined
> var y = 20
undefined
> x + y
30
> var sum = _
undefined
> console.log(sum)
30
undefined
```

```
>
```

Node.js - npm

npm stands for Node Package Manager. npm provides following two main functionalities:

- Online repositories for node.js packages/modules which are searchable on search.nodejs.org
- Command line utility to install 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:

```
C:\Nodejs_WorkSpace>npm --version
2.5.1
```

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().

Globally installed packages/dependencies are stored in **<user-directory>**/npm 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.

Let's install express, a popular web framework using local installation.

```
C:\Nodejs_WorkSpace>npm install express
express@4.11.2 node_modules\express
|-- merge-descriptors@0.0.2
|-- utils-merge@1.0.0
|-- methods@1.1.1
|-- escape-html@1.0.1
|-- fresh@0.2.4
|-- cookie@0.1.2
|-- range-parser@1.0.2
|-- media-typer@0.3.0
|-- cookie-signature@1.0.5
|-- vary@1.0.0
|-- finalhandler@0.3.3
|-- parseurl@1.3.0
```

```
|-- serve-static@1.8.1
|-- content-disposition@0.5.0
|-- path-to-regexp@0.1.3
|-- depd@1.0.0
|-- qs@2.3.3
|-- debug@2.1.1 (ms@0.6.2)
|-- send@0.11.1 (destroy@1.0.3, ms@0.7.0, mime@1.2.11)
|-- on-finished@2.2.0 (ee-first@1.1.0)
|-- type-is@1.5.7 (mime-types@2.0.9)
|-- accepts@1.2.3 (negotiator@0.5.0, mime-types@2.0.9)
|-- etag@1.5.1 (crc@3.2.1)
|-- proxy-addr@1.0.6 (forwarded@0.1.0, ipaddr.js@0.1.8)
```

Once npm completes the download, you can verify by looking at the content of C:\Nodejs_WorkSpace\node_modules. Or type the following command:

```
C:\Nodejs_WorkSpace>npm ls
C:\Nodejs_WorkSpace
|-- express@4.11.2
  |-- accepts@1.2.3
  | |-- mime-types@2.0.9
  | | |-- mime-db@1.7.0
  | |-- negotiator@0.5.0
  |-- content-disposition@0.5.0
  |-- cookie@0.1.2
  |-- cookie-signature@1.0.5
  |-- debug@2.1.1
  | |-- ms@0.6.2
  |-- depd@1.0.0
  |-- escape-html@1.0.1
  |-- etag@1.5.1
  | |-- crc@3.2.1
  |-- finalhandler@0.3.3
  |-- fresh@0.2.4
  |-- media-typer@0.3.0
  |-- merge-descriptors@0.0.2
  |-- methods@1.1.1
  |-- on-finished@2.2.0
  | |-- ee-first@1.1.0
  |-- parseurl@1.3.0
  |-- path-to-regexp@0.1.3
  |-- proxy-addr@1.0.6
  | |-- forwarded@0.1.0
  | |-- ipaddr.js@0.1.8
  |-- qs@2.3.3
```

```
  |-- range-parser@1.0.2
  |-- send@0.11.1
  | |-- destroy@1.0.3
  | |-- mime@1.2.11
  | |-- ms@0.7.0
  |-- serve-static@1.8.1
  |-- type-is@1.5.7
  | |-- mime-types@2.0.9
  |   |-- mime-db@1.7.0
  |-- utils-merge@1.0.0
  |-- vary@1.0.0
```

Now Let's try installing express, a popular web framework using global installation.

```
C:\Nodejs_WorkSpace>npm install express - g
```

Once npm completes the download, you can verify by looking at the content of **<user-directory>**/npm/node_modules. Or type the following command:

```
C:\Nodejs_WorkSpace>npm ls -g
```

Installing a module

Installation of any module is as simple as typing the following command.

```
C:\Nodejs_WorkSpace>npm install express
```

Now you can use it in your js file as following:

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

Using package.json

package.json is present in the root directoryt of any Node application/module and is used to define the properties of a package. Let's open package.json of express package present in C:\Nodejs_Workspace\node_modules\express\

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

```
    {
      "name": "Ciaran Jessup",
      "email": "ciaranj@gmail.com"
    },
    {
      "name": "Douglas Christopher Wilson",
      "email": "doug@somethingdoug.com"
    },
    {
      "name": "Guillermo Rauch",
      "email": "rauchg@gmail.com"
    },
    {
      "name": "Jonathan Ong",
      "email": "me@jongleberry.com"
    },
    {
      "name": "Roman Shtylman",
      "email": "shtylman+expressjs@gmail.com"
    },
    {
      "name": "Young Jae Sim",
      "email": "hanul@hanul.me"
    }
  ],
  "license": "MIT",
  "repository": {
    "type": "git",
    "url": "https://github.com/strongloop/express"
  },
  "homepage": "http://expressjs.com/",
  "keywords": [
    "express",
    "framework",
    "sinatra",
    "web",
    "rest",
    "restful",
    "router",
    "app",
    "api"
  ],
  "dependencies": {
    "accepts": "~1.2.3",
    "content-disposition": "0.5.0",
    "cookie-signature": "1.0.5",
    "debug": "~2.1.1",
    "depd": "~1.0.0",
    "escape-html": "1.0.1",
    "etag": "~1.5.1",
    "finalhandler": "0.3.3",
    "fresh": "0.2.4",
    "media-typer": "0.3.0",
    "methods": "~1.1.1",
    "on-finished": "~2.2.0",
    "parseurl": "~1.3.0",
    "path-to-regexp": "0.1.3",
    "proxy-addr": "~1.0.6",
    "qs": "2.3.3",
    "range-parser": "~1.0.2",
    "send": "0.11.1",
```

```
    "serve-static": "~1.8.1",
    "type-is": "~1.5.6",
    "vary": "~1.0.0",
    "cookie": "0.1.2",
    "merge-descriptors": "0.0.2",
    "utils-merge": "1.0.0"
  },
  "devDependencies": {
    "after": "0.8.1",
    "ejs": "2.1.4",
    "istanbul": "0.3.5",
    "marked": "0.3.3",
    "mocha": "~2.1.0",
    "should": "~4.6.2",
    "supertest": "~0.15.0",
    "hjs": "~0.0.6",
    "body-parser": "~1.11.0",
    "connect-redis": "~2.2.0",
    "cookie-parser": "~1.3.3",
    "express-session": "~1.10.2",
    "jade": "~1.9.1",
    "method-override": "~2.3.1",
    "morgan": "~1.5.1",
    "multiparty": "~4.1.1",
    "vhost": "~3.0.0"
  },
  "engines": {
    "node": ">= 0.10.0"
  },
  "files": [
    "LICENSE",
    "History.md",
    "Readme.md",
    "index.js",
    "lib/"
  ],
  "scripts": {
    "test": "mocha --require test/support/env --reporter spec --bail --check-leaks test/ test/a
    "test-cov": "istanbul cover node_modules/mocha/bin/_mocha -- --require test/support/env --r
    "test-tap": "mocha --require test/support/env --reporter tap --check-leaks test/ test/accep
    "test-travis": "istanbul cover node_modules/mocha/bin/_mocha --report lcovonly -- --require
  },
  "gitHead": "63ab25579bda70b4927a179b580a9c580b6c7ada",
  "bugs": {
    "url": "https://github.com/strongloop/express/issues"
  },
  "_id": "express@4.11.2",
  "_shasum": "8df3d5a9ac848585f00a0777601823faecd3b148",
  "_from": "express@*",
  "_npmVersion": "1.4.28",
  "_npmUser": {
    "name": "dougwilson",
    "email": "doug@somethingdoug.com"
  },
  "maintainers": [
    {
      "name": "tjholowaychuk",
      "email": "tj@vision-media.ca"
    },
    {
      "name": "jongleberry",
```

```
      "email": "jonathanrichardong@gmail.com"
    },
    {
      "name": "shtylman",
      "email": "shtylman@gmail.com"
    },
    {
      "name": "dougwilson",
      "email": "doug@somethingdoug.com"
    },
    {
      "name": "aredridel",
      "email": "aredridel@nbtsc.org"
    },
    {
      "name": "strongloop",
      "email": "callback@strongloop.com"
    },
    {
      "name": "rfeng",
      "email": "enjoyjava@gmail.com"
    }
  ],
  "dist": {
    "shasum": "8df3d5a9ac848585f00a0777601823faecd3b148",
    "tarball": "http://registry.npmjs.org/express/-/express-4.11.2.tgz"
  },
  "directories": {},
  "_resolved": "https://registry.npmjs.org/express/-/express-4.11.2.tgz",
  "readme": "ERROR: No README data found!"
}
```


Attributes of Package.json

name - name of the package

version - version of the package

description - description of the package

homepage - homepage of the package

author - author of the package

contributors - name of the contributors to the package

dependencies - list of dependencies. npm automatically installs all the dependencies mentioned here in the node_module folder of the package.

repository - repository type and url of the package

main - entry point of the package

keywords - keywords

Uninstalling a module

Use following command to uninstall a module.

```
C:\Nodejs_WorkSpace>npm uninstall express
```

Once npm uninstall the package, you can verify by looking at the content of **<user-directory>**/npm/node_modules. Or type the following command:

```
C:\Nodejs_WorkSpace>npm ls
```

Updating a module

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

```
C:\Nodejs_WorkSpace>npm update
```

Search a module

Search package name using npm.

```
C:\Nodejs_WorkSpace>npm search express
```

Create a module

Creation of module requires package.json to be generated. Let's generate package.json using npm.

```
C:\Nodejs_WorkSpace>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: (Nodejs_WorkSpace)
```

Once package.json is generated. Use following command to register yourself with

npm repository site using a valid email address.

```
C:\Nodejs_WorkSpace>npm adduser
```

Now its time to publish your module:

```
C:\Nodejs_WorkSpace>npm publish
```

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 txt file named test.txt in **C:\>Nodejs_WorkSpace**

```
TutorialsPoint.Com
```

Create a js file named test.js in **C:\>Nodejs_WorkSpace**

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

Now run the test.js to see the result:

```
C:\Nodejs_WorkSpace>node test.js
```

Verify the Output

```
TutorialsPoint.Com
Program Ended
```

Non-Blocking Code Example

Create a txt file named test.txt in **C:\>Nodejs_WorkSpace**

```
TutorialsPoint.Com
```

Update test.js in **C:\>Nodejs_WorkSpace**

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

fs.readFile('test.txt', function (err, data) {
    if (err) return console.error(err);
    console.log(data.toString());
});
console.log("Program Ended");
```

Now run the test.js to see the result:

```
C:\Nodejs_WorkSpace>node test.js
```

Verify the Output

```
Program Ended
TutorialsPoint.Com
```

Event Loop Overview

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 reason 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.

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 event handling works on the observer pattern. The functions which listens to events acts as observer. Whenever an event got fired, its listener function starts executing. Node.js has multiple in-built event. The primary actor is EventEmitter which can be imported using following syntax

```
//import events module
var events = require('events');
//create an eventEmitter object
var eventEmitter = new events.EventEmitter();
```

Example

Create a js file named test.js in **C:\>Nodejs_WorkSpace**.

File: test.js

```
//import events module
var events = require('events');
//create an eventEmitter object
var eventEmitter = new events.EventEmitter();

//create a function connected which is to be executed
//when 'connection' event occurs
var connected = function connected() {
   console.log('connection succesful.');

   // fire the data_received event
   eventEmitter.emit('data_received.');
}

// bind the connection event with the connected function
eventEmitter.on('connection', connected);

// 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 run the test.js to see the result:

```
C:\Nodejs_WorkSpace>node test.js
```

Verify the Output. Server has started

```
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.

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

fs.readFile('test.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 occur 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.

Node.js - Event Emitter

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

Sr. No.	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

Sr. No.	method	Description
	listenerCount(emitter,	Return the number of listeners for a given

1	**event)**	event.

Events

Sr. No.	event name	Parameters	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 test.js in **C:\>Nodejs_WorkSpace**.

File: test.js

```
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 test.js to see the result:

```
C:\Nodejs_WorkSpace>node test.js
```

Verify the Output. Server has started

```
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 - Buffer Module

buffer module can be used to create Buffer and SlowBuffer classes. Buffer module can be imported using following syntax.

```
var buffer = require("buffer")
```

Buffer class

Buffer class is a global class and can be accessed in application without importing buffer module. A Buffer is a kind of an array of integers and corresponds to a raw memory allocation outside the V8 heap. A Buffer cannot be resized.

Methods

Sr. No.	method	Parameters	Description
1	**new Buffer(size)**	size Number	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)**	buffer Buffer	Copies the passed buffer data onto a new Buffer instance.
3	**new Buffer(str[, encoding])**	str String - string to encode. encoding String - encoding to use, Optional.	Allocates a new buffer containing the given str. encoding defaults to 'utf8'.
4	**buf.length**	Return: Number	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])**		string String - data to be written to buffer offset Number, Optional, Default: 0 length Number, Optional, Default: buffer.length - offset encoding String, Optional, Default: 'utf8'	Allocates a new buffer containing the given str. encoding defaults to 'utf8'.
6	**buf.writeUIntLE(value, offset, byteLength[, noAssert])**		value {Number} Bytes to be written to buffer offset {Number} 0 <= offset <= buf.length byteLength {Number} 0 < byteLength <= 6 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.

			{Boolean} Default: false Return: {Number}	
7	**buf.writeUIntBE(value, offset, byteLength[, noAssert])**		value {Number} Bytes to be written to buffer offset {Number} 0 <= offset <= buf.length byteLength {Number} 0 < byteLength <= 6 noAssert {Boolean} Default: false Return: {Number}	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.
			value {Number} Bytes to be written to buffer offset {Number}	

8	**buf.writeIntLE(value, offset, byteLength[, noAssert])**		0 <= offset <= buf.length byteLength {Number} 0 < byteLength <= 6 noAssert {Boolean} Default: false Return: {Number}	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.
9	**buf.writeIntBE(value, offset, byteLength[, noAssert])**		value {Number} Bytes to be written to buffer offset {Number} 0 <= offset <= buf.length byteLength {Number} 0 < byteLength <= 6 noAssert {Boolean} Default: false Return:	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.

		{Number}	
10	**buf.readUIntLE(offset, byteLength[, noAssert])**	offset {Number} 0 <= offset <= buf.length byteLength {Number} 0 < byteLength <= 6 noAssert {Boolean} Default: false Return: {Number}	A generalized version of all numeric read methods. Supports up to 48 bits of accuracy.Set noAssert to true to skip validation of offset. This means that offset may be beyond the end of the buffer. Defaults to false.
11	**buf.readUIntBE(offset, byteLength[, noAssert])**	offset {Number} 0 <= offset <= buf.length byteLength {Number} 0 < byteLength <= 6 noAssert {Boolean} Default: false Return: {Number}	A generalized version of all numeric read methods. Supports up to 48 bits of accuracy.Set noAssert to true to skip validation of offset. This means that offset may be beyond the end of the buffer. Defaults to false.

12	**buf.readIntLE(offset, byteLength[, noAssert])**	offset {Number} 0 <= offset <= buf.length byteLength {Number} 0 < byteLength <= 6 noAssert {Boolean} Default: false Return: {Number}	A generalized version of all numeric read methods. Supports up to 48 bits of accuracy.Set noAssert to true to skip validation of offset. This means that offset may be beyond the end of the buffer. Defaults to false.
13	**buf.readIntBE(offset, byteLength[, noAssert])**	offset {Number} 0 <= offset <= buf.length byteLength {Number} 0 < byteLength <= 6 noAssert {Boolean} Default: false Return: {Number}	A generalized version of all numeric read methods. Supports up to 48 bits of accuracy.Set noAssert to true to skip validation of offset. This means that offset may be beyond the end of the buffer. Defaults to false.

14	**buf.toString([encoding][, start][, end])**		encoding String, Optional, Default: 'utf8' start Number, Optional, Default: 0 end Number, Optional, Default: buffer.length	Decodes and returns a string from buffer data encoded using the specified character set encoding.
15	**buf.toJSON()**			Returns a JSON-representation of the Buffer instance. JSON.stringify implicitly calls this function when stringifying a Buffer instance.
16	**buf[index]**			Get and set the octet at index. The values refer to individual bytes, so the legal range is between 0x00 and 0xFF hex or 0 and 255.
17	**buf.equals(otherBuffer)**		otherBuffer Buffer	Returns a boolean of whether this and otherBuffer have the same bytes.
18	**buf.compare(otherBuffer)**		otherBuffer Buffer	Returns a number indicating whether this comes before or after or is the same as the otherBuffer in sort order.
			targetBuffer	

19	**buf.copy(targetBuffer[, targetStart][, sourceStart][, sourceEnd])**		Buffer object - Buffer to copy into targetStart Number, Optional, Default: 0 sourceStart Number, Optional, Default: 0 sourceEnd Number, Optional, Default: buffer.length	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.
20	**buf.slice([start][, end])**		start Number, Optional, Default: 0 end Number, Optional, Default: buffer.length	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.
21	**buf.readUInt8(offset[, noAssert])**		offset Number noAssert Boolean, Optional, Default: false	Reads an unsigned 8 bit integer from the buffer at the specified offset. Set noAssert to true to skip validation of offset. This means that offset may be beyond the end of the

			Return: Number	buffer. Defaults to false.
22	**buf.readUInt16LE(offset[, noAssert])**		offset Number noAssert Boolean, Optional, Default: false Return: Number	Reads an unsigned 16 bit integer from the buffer at the specified offset with specified endian format. Set noAssert to true to skip validation of offset. This means that offset may be beyond the end of the buffer. Defaults to false.
23	**buf.readUInt16BE(offset[, noAssert])**		offset Number noAssert Boolean, Optional, Default: false Return: Number	Reads an unsigned 16 bit integer from the buffer at the specified offset with specified endian format. Set noAssert to true to skip validation of offset. This means that offset may be beyond the end of the buffer. Defaults to false.
24	**buf.readUInt32LE(offset[, noAssert])**		offset Number noAssert Boolean, Optional, Default: false Return: Number	Reads an unsigned 32 bit integer from the buffer at the specified offset with specified endian format. Set noAssert to true to skip validation of offset. This means that offset may be beyond the end of the buffer. Defaults to false.

25	**buf.readUInt32BE(offset[, noAssert])**		offset Number noAssert Boolean, Optional, Default: false Return: Number	Reads an unsigned 32 bit integer from the buffer at the specified offset with specified endian format. Set noAssert to true to skip validation of offset. This means that offset may be beyond the end of the buffer. Defaults to false.
26	**buf.readInt8(offset[, noAssert])**		offset Number noAssert Boolean, Optional, Default: false Return: Number	Reads a signed 8 bit integer from the buffer at the specified offset. Set noAssert to true to skip validation of offset. This means that offset may be beyond the end of the buffer. Defaults to false.
27	**buf.readInt16LE(offset[, noAssert])**		offset Number noAssert Boolean, Optional, Default: false Return: Number	Reads a signed 16 bit integer from the buffer at the specified offset with specified endian format. Set noAssert to true to skip validation of offset. This means that offset may be beyond the end of the buffer. Defaults to false.
			offset Number noAssert Boolean,	Reads a signed 16 bit integer from the buffer at the specified offset with specified endian format.

28	**buf.readInt16BE(offset[, noAssert])**		Optional, Default: false Return: Number	Set noAssert to true to skip validation of offset. This means that offset may be beyond the end of the buffer. Defaults to false.
29	**buf.readInt32LE(offset[, noAssert])**		offset Number noAssert Boolean, Optional, Default: false Return: Number	Reads a signed 32 bit integer from the buffer at the specified offset with specified endian format. Set noAssert to true to skip validation of offset. This means that offset may be beyond the end of the buffer. Defaults to false.
30	**buf.readInt32BE(offset[, noAssert])**		offset Number noAssert Boolean, Optional, Default: false Return: Number	Reads a signed 32 bit integer from the buffer at the specified offset with specified endian format. Set noAssert to true to skip validation of offset. This means that offset may be beyond the end of the buffer. Defaults to false.
31	**buf.readFloatLE(offset[, noAssert])**		offset Number noAssert Boolean, Optional, Default: false	Reads a 32 bit float from the buffer at the specified offset with specified endian format. Set noAssert to true to skip validation of offset. This means that offset may be beyond the end of the buffer. Defaults

			Return: Number	to false.
32	**buf.readFloatBE(offset[, noAssert])**		offset Number noAssert Boolean, Optional, Default: false Return: Number	Reads a 32 bit float from the buffer at the specified offset with specified endian format. Set noAssert to true to skip validation of offset. This means that offset may be beyond the end of the buffer. Defaults to false.
33	**buf.readDoubleLE(offset[, noAssert])**		offset Number noAssert Boolean, Optional, Default: false Return: Number	Reads a 64 bit double from the buffer at the specified offset with specified endian format. Set noAssert to true to skip validation of offset. This means that offset may be beyond the end of the buffer. Defaults to false.
34	**buf.readDoubleBE(offset[, noAssert])**		offset Number noAssert Boolean, Optional, Default: false Return: Number	Reads a 64 bit double from the buffer at the specified offset with specified endian format. Set noAssert to true to skip validation of offset. This means that offset may be beyond the end of the buffer. Defaults to false.
				Writes value to the buffer at the specified offset.

35	**buf.writeUInt8(value, offset[, noAssert])**		value Number offset Number noAssert Boolean, Optional, Default: false	Note, value must be a valid unsigned 8 bit integer. Set noAssert to true to skip validation of value and offset. This means that value may be too large for the specific function and offset may be beyond the end of the buffer leading to the values being silently dropped. This should not be used unless you are certain of correctness. Defaults to false.
36	**buf.writeUInt16LE(value, offset[, noAssert])**		value Number offset Number noAssert Boolean, Optional, Default: false	Writes value to the buffer at the specified offset with specified endian format. Note, value must be a valid unsigned 16 bit integer. Set noAssert to true to skip validation of value and offset. This means that value may be too large for the specific function and offset may be beyond the end of the buffer leading to the values being silently dropped. This should not be used unless you are certain of correctness. Defaults to false.
37	**buf.writeUInt16BE(value, offset[, noAssert])**		value Number offset Number noAssert Boolean, Optional,	Writes value to the buffer at the specified offset with specified endian format. Note, value must be a valid unsigned 16 bit integer. Set noAssert to true to skip validation of value and offset. This means that value may be too large for the specific function and offset may be beyond the

			Default: false	end of the buffer leading to the values being silently dropped. This should not be used unless you are certain of correctness. Defaults to false.
38	**buf.writeUInt32LE(value, offset[, noAssert])**		value Number offset Number noAssert Boolean, Optional, Default: false	Writes value to the buffer at the specified offset with specified endian format. Note, value must be a valid unsigned 32 bit integer. Set noAssert to true to skip validation of value and offset. This means that value may be too large for the specific function and offset may be beyond the end of the buffer leading to the values being silently dropped. This should not be used unless you are certain of correctness. Defaults to false.
39	**buf.writeUInt32BE(value, offset[, noAssert])**		value Number offset Number noAssert Boolean, Optional, Default: false	Writes value to the buffer at the specified offset with specified endian format. Note, value must be a valid unsigned 32 bit integer. Set noAssert to true to skip validation of value and offset. This means that value may be too large for the specific function and offset may be beyond the end of the buffer leading to the values being silently dropped. This should not be used unless you are certain of correctness. Defaults to false.
				Writes value to the buffer at the specified offset with specified endian format.

40	**buf.writeInt8(value, offset[, noAssert])**		value Number offset Number noAssert Boolean, Optional, Default: false	Note, value must be a valid signed 8 bit integer. Set noAssert to true to skip validation of value and offset. This means that value may be too large for the specific function and offset may be beyond the end of the buffer leading to the values being silently dropped. This should not be used unless you are certain of correctness. Defaults to false.
41	**buf.writeInt16LE(value, offset[, noAssert])**		value Number offset Number noAssert Boolean, Optional, Default: false	Writes value to the buffer at the specified offset with specified endian format. Note, value must be a valid signed 16 bit integer. Set noAssert to true to skip validation of value and offset. This means that value may be too large for the specific function and offset may be beyond the end of the buffer leading to the values being silently dropped. This should not be used unless you are certain of correctness. Defaults to false.
42	**buf.writeInt16BE(value, offset[, noAssert])**		value Number offset Number noAssert Boolean, Optional,	Writes value to the buffer at the specified offset with specified endian format. Note, value must be a valid signed 16 bit integer. Set noAssert to true to skip validation of value and offset. This means that value may be too large for the specific function and offset may be beyond the

			Default: false	end of the buffer leading to the values being silently dropped. This should not be used unless you are certain of correctness. Defaults to false.
43	**buf.writeInt32LE(value, offset[, noAssert])**		value Number offset Number noAssert Boolean, Optional, Default: false	Writes value to the buffer at the specified offset with specified endian format. Note, value must be a valid signed 32 bit integer. Set noAssert to true to skip validation of value and offset. This means that value may be too large for the specific function and offset may be beyond the end of the buffer leading to the values being silently dropped. This should not be used unless you are certain of correctness. Defaults to false.
44	**buf.writeInt32BE(value, offset[, noAssert])**		value Number offset Number noAssert Boolean, Optional, Default: false	Writes value to the buffer at the specified offset with specified endian format. Note, value must be a valid signed 32 bit integer. Set noAssert to true to skip validation of value and offset. This means that value may be too large for the specific function and offset may be beyond the end of the buffer leading to the values being silently dropped. This should not be used unless you are certain of correctness. Defaults to false.
				Writes value to the buffer at the specified offset with specified endian format.

45	**buf.writeFloatLE(value, offset[, noAssert])**		value Number offset Number noAssert Boolean, Optional, Default: false	Note, value must be a valid 32 bit float. Set noAssert to true to skip validation of value and offset. This means that value may be too large for the specific function and offset may be beyond the end of the buffer leading to the values being silently dropped. This should not be used unless you are certain of correctness. Defaults to false.
46	**buf.writeFloatBE(value, offset[, noAssert])**		value Number offset Number noAssert Boolean, Optional, Default: false	Writes value to the buffer at the specified offset with specified endian format. Note, value must be a valid 32 bit float. Set noAssert to true to skip validation of value and offset. This means that value may be too large for the specific function and offset may be beyond the end of the buffer leading to the values being silently dropped. This should not be used unless you are certain of correctness. Defaults to false.
47	**buf.writeDoubleLE(value, offset[, noAssert])**		value Number offset Number noAssert Boolean, Optional,	Writes value to the buffer at the specified offset with specified endian format. Note, value must be a valid 64 bit double. Set noAssert to true to skip validation of value and offset. This means that value may be too large for the specific function and offset may be beyond the end of the buffer leading

			Default: false	to the values being silently dropped. This should not be used unless you are certain of correctness. Defaults to false.
48	**buf.writeDoubleBE(value, offset[, noAssert])**		value Number offset Number noAssert Boolean, Optional, Default: false	Writes value to the buffer at the specified offset with specified endian format. Note, value must be a valid 64 bit double. Set noAssert to true to skip validation of value and offset. This means that value may be too large for the specific function and offset may be beyond the end of the buffer leading to the values being silently dropped. This should not be used unless you are certain of correctness. Defaults to false.
49	**buf.fill(value[, offset][, end])**		value Number offset Number, Optional end Number, Optional	Fills the buffer with the specified value. If the offset (defaults to 0) and end (defaults to buffer.length) are not given it will fill the entire buffer.

Class Methods

Sr. No.	method	Parameters	Description
1	**Buffer.isEncoding(encoding)**	encoding String The encoding	Returns true if the encoding is a valid encoding argument, or

			string to test	false otherwise.
2	**Buffer.isBuffer(obj)**		obj Object Return: Boolean	Tests if obj is a Buffer.
3	**Buffer.byteLength(string[, encoding])**		string String encoding String, Optional, Default: 'utf8' Return: Number	Gives the actual byte length of a string. encoding defaults to 'utf8'. This is not the same as String.prototype.length since that returns the number of characters in a string.
4	**Buffer.concat(list[, totalLength])**		list Array List of Buffer objects to concat totalLength Number Total length of the buffers when concatenated	Returns a buffer which is the result of concatenating all the buffers in the list together.
5	**Buffer.compare(buf1, buf2)**		buf1 Buffer buf2 Buffer	The same as buf1.compare(buf2). Useful for sorting an Array of Buffers.

Example

Create a js file named test.js in **C:\>Nodejs_WorkSpace**.

File: test.js

```
//create a buffer
var buffer = new Buffer(26);
console.log("buffer length: " + buffer.length);

//write to buffer
var data = "TutorialsPoint.com";
buffer.write(data);
console.log(data + ": " + data.length + " characters, " + Buffer.byteLength(data, 'utf8') + " b

//slicing a buffer
var buffer1 = buffer.slice(0,14);
console.log("buffer1 length: " + buffer1.length);
console.log("buffer1 content: " + buffer1.toString());

//modify buffer by indexes
for (var i = 0 ; i < 26 ; i++) {
  buffer[i] = i + 97; // 97 is ASCII a
}
console.log("buffer content: " + buffer.toString('ascii'));

var buffer2 = new Buffer(4);

buffer2[0] = 0x3;
buffer2[1] = 0x4;
buffer2[2] = 0x23;
buffer2[3] = 0x42;

//reading from buffer
console.log(buffer2.readUInt16BE(0));
console.log(buffer2.readUInt16LE(0));
console.log(buffer2.readUInt16BE(1));
console.log(buffer2.readUInt16LE(1));
console.log(buffer2.readUInt16BE(2));
console.log(buffer2.readUInt16LE(2));


var buffer3 = new Buffer(4);
buffer3.writeUInt16BE(0xdead, 0);
buffer3.writeUInt16BE(0xbeef, 2);

console.log(buffer3);

buffer3.writeUInt16LE(0xdead, 0);
buffer3.writeUInt16LE(0xbeef, 2);

console.log(buffer3);
//convert to a JSON Object
var json = buffer3.toJSON();
console.log("JSON Representation : ");
console.log(json);

//Get a buffer from JSON Object
var buffer6 = new Buffer(json);
console.log(buffer6);

//copy a buffer
var buffer4 = new Buffer(26);
buffer.copy(buffer4);
console.log("buffer4 content: " + buffer4.toString());

//concatenate a buffer
```

```
var buffer5 = Buffer.concat([buffer,buffer4]);
console.log("buffer5 content: " + buffer5.toString());
```

Now run the test.js to see the result:

```
C:\Nodejs_WorkSpace>node test.js
```

Verify the Output.

```
buffer length: 26
TutorialsPoint.com: 18 characters, 18 bytes
buffer1 length: 14
buffer1 content: TutorialsPoint
buffer content: abcdefghijklmnopqrstuvwxyz
772
1027
1059
8964
9026
16931
<Buffer de ad be ef>
<Buffer ad de ef be>
buffer4 content: abcdefghijklmnopqrstuvwxyz
buffer5 content: abcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyz
```

Node.js - Streams

What are Streams?

Streams are objects that let yu read data from a source or write data to destination in continous fashion. In Node, 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 and throws several events at 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

Reading from stream

Create a txt file named test.txt in **C:\>Nodejs_WorkSpace**

```
TutorialsPoint.Com
```

Create test.js in **C:\>Nodejs_WorkSpace**

```
var fs = require("fs");
var data = '';
//create a readable stream
var readerStream = fs.createReadStream('test.txt');

//set the encoding to be utf8.
readerStream.setEncoding('UTF8');

//handle stream events
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 test.js to see the result:

```
C:\Nodejs_WorkSpace>node test.js
```

Verify the Output

```
Program Ended
TutorialsPoint.Com
```

Writing to stream

Update test.js in **C:\>Nodejs_WorkSpace**

```
var fs = require("fs");
var data = 'TutorialsPoint.Com';
//create a writable stream
var writerStream = fs.createWriteStream('test1.txt');

//write the data to stream
//set the encoding to be utf8.
writerStream.write(data,'UTF8');

//mark the end of file
writerStream.end();

//handle stream events
writerStream.on('finish', function() {
    console.log("Write completed.");
});

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

Now run the test.js to see the result:

```
C:\Nodejs_WorkSpace>node test.js
```

Verify the Output

```
Program Ended
Write completed.
```

Open test1.txt in **C:\>Nodejs_WorkSpace**.Verify the result.

```
TutorialsPoint.Com
```

Piping streams

Piping is a mechanism to connect output of one stream 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. Consider the above example, where we've read test.txt using readerStream and write test1.txt using writerStream. Now we'll use the piping to simplify our operation or reading from one file and writing to another file.

Update test.js in **C:\>Nodejs_WorkSpace**

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

//create a readable stream
var readerStream = fs.createReadStream('test.txt');

//create a writable stream
var writerStream = fs.createWriteStream('test2.txt');

//pipe the read and write operations
//read test.txt and write data to test2.txt
readerStream.pipe(writerStream);

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

Now run the test.js to see the result:

```
C:\Nodejs_WorkSpace>node test.js
```

Verify the Output

```
Program Ended
```

Open test2.txt in **C:\>Nodejs_WorkSpace**. Verify the result.

```
TutorialsPoint.Com
```

Node.js - File System

fs module is used for File I/O. 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 the latter one does.

Example

Create a js file named test.js in **C:\>Nodejs_WorkSpace**.

File: test.js

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

//Asynchronous read
fs.readFile('test.txt', function (err, data) {
    if (err) return console.error(err);
    console.log("Asynchronous read: " + data.toString());
});

//Synchronous read
var data = fs.readFileSync('test.txt');
console.log("Synchronous read: " + data.toString());

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

Now run the test.js to see the result:

```
C:\Nodejs_WorkSpace>node test.js
```

Verify the Output.

```
Synchronous read: TutorialsPoint.Com
Program Ended
Asynchronous read: TutorialsPoint.Com
```

Methods

Sr. No.	method	Description
1	**fs.rename(oldPath, newPath, callback)**	Asynchronous rename(). No arguments other than a possible exception are given to the completion callback.
2	**fs.ftruncate(fd, len, callback)**	Asynchronous ftruncate(). No arguments other than a possible exception are given to the completion callback.
3	**fs.ftruncateSync(fd, len)**	Synchronous ftruncate()
4	**fs.truncate(path, len, callback)**	Asynchronous truncate(). No arguments other than a possible exception are given to the completion callback.
5	**fs.truncateSync(path, len)**	Synchronous truncate()
6	**fs.chown(path, uid, gid, callback)**	Asynchronous chown(). No arguments other than a possible exception are given to the completion callback.
7	**fs.chownSync(path, uid, gid)**	Synchronous chown()

8	**fs.fchown(fd, uid, gid, callback)**	Asynchronous fchown(). No arguments other than a possible exception are given to the completion callback.
9	**fs.fchownSync(fd, uid, gid)**	Synchronous fchown()
10	**fs.lchown(path, uid, gid, callback)**	Asynchronous lchown(). No arguments other than a possible exception are given to the completion callback.
11	**fs.lchownSync(path, uid, gid)**	Synchronous lchown()
12	**fs.chmod(path, mode, callback)**	Asynchronous chmod(). No arguments other than a possible exception are given to the completion callback.
13	**fs.chmodSync(path, mode)**	Synchronous chmod().
14	**fs.fchmod(fd, mode, callback)**	Asynchronous fchmod(). No arguments other than a possible exception are given to the completion callback.
15	**fs.fchmodSync(fd, mode)**	Synchronous fchmod().
16	**fs.lchmod(path, mode, callback)**	Asynchronous lchmod(). No arguments other than a possible exception are given to the completion callback.Only available on Mac OS X.
17	**fs.lchmodSync(path, mode)**	Synchronous lchmod().
18	**fs.stat(path, callback)**	Asynchronous stat(). The callback gets two arguments (err, stats) where stats is a fs.Stats object.
19	**fs.lstat(path, callback)**	Asynchronous lstat(). The callback gets two arguments (err, stats) where stats is a fs.Stats object. lstat() is identical to stat(), except that if path is a symbolic link, then the link itself is stat-ed, not the file that it refers to.
20	**fs.fstat(fd, callback)**	Asynchronous fstat(). The callback gets two arguments (err, stats) where stats is a fs.Stats object. fstat() is identical to stat(), except that the file to be stat-ed is specified by the file descriptor fd.
21	**fs.statSync(path)**	Synchronous stat(). Returns an instance of fs.Stats.
22	**fs.lstatSync(path)**	Synchronous lstat(). Returns an instance of

		fs.Stats.
23	**fs.fstatSync(fd)**	Synchronous fstat(). Returns an instance of fs.Stats.
24	**fs.link(srcpath, dstpath, callback)**	Asynchronous link(). No arguments other than a possible exception are given to the completion callback.
25	**fs.linkSync(srcpath, dstpath)**	Synchronous link().
26	**fs.symlink(srcpath, dstpath[, type], callback)**	Asynchronous symlink(). No arguments other than a possible exception are given to the completion callback. The type argument can be set to 'dir', 'file', or 'junction' (default is 'file') and is only available on Windows (ignored on other platforms). Note that Windows junction points require the destination path to be absolute. When using 'junction', the destination argument will automatically be normalized to absolute path.
27	**fs.symlinkSync(srcpath, dstpath[, type])**	Synchronous symlink().
28	**fs.readlink(path, callback)**	Asynchronous readlink(). The callback gets two arguments (err, linkString).
29	**fs.realpath(path[, cache], callback)**	Asynchronous realpath(). The callback gets two arguments (err, resolvedPath). May use process.cwd to resolve relative paths. cache is an object literal of mapped paths that can be used to force a specific path resolution or avoid additional fs.stat calls for known real paths.
30	**fs.realpathSync(path[, cache])**	Synchronous realpath(). Returns the resolved path.
31	**fs.unlink(path, callback)**	Asynchronous unlink(). No arguments other than a possible exception are given to the completion callback.
32	**fs.unlinkSync(path)**	Synchronous unlink().
33	**fs.rmdir(path, callback)**	Asynchronous rmdir(). No arguments other than a possible exception are given to the completion callback.
34	**fs.rmdirSync(path)**	Synchronous rmdir().

35	**fs.mkdir(path[, mode], callback)**	SAsynchronous mkdir(2). No arguments other than a possible exception are given to the completion callback. mode defaults to 0777.
36	**fs.mkdirSync(path[, mode])**	Synchronous mkdir().
37	**fs.readdir(path, callback)**	Asynchronous readdir(3). Reads the contents of a directory. The callback gets two arguments (err, files) where files is an array of the names of the files in the directory excluding '.' and '..'.
38	**fs.readdirSync(path)**	Synchronous readdir(). Returns an array of filenames excluding '.' and '..'.
39	**fs.close(fd, callback)**	Asynchronous close(). No arguments other than a possible exception are given to the completion callback.
40	**fs.closeSync(fd)**	Synchronous close().
41	**fs.open(path, flags[, mode], callback)**	Asynchronous file open.
42	**fs.openSync(path, flags[, mode])**	Synchronous version of fs.open().
43	**fs.utimes(path, atime, mtime, callback)**	
44	**fs.utimesSync(path, atime, mtime)**	Change file timestamps of the file referenced by the supplied path.
45	**fs.futimes(fd, atime, mtime, callback)**	
46	**fs.futimesSync(fd, atime, mtime)**	Change the file timestamps of a file referenced by the supplied file descriptor.
47	**fs.fsync(fd, callback)**	Asynchronous fsync(2). No arguments other than a possible exception are given to the completion callback.
48	**fs.fsyncSync(fd)**	Synchronous fsync(2).
49	**fs.write(fd, buffer, offset, length[, position], callback)**	Write buffer to the file specified by fd.
50	**fs.write(fd, data[, position[, encoding]], callback)**	Write data to the file specified by fd. If data is not a Buffer instance then the value will be coerced to a string.

51	**fs.writeSync(fd, buffer, offset, length[, position])**	Synchronous versions of fs.write(). Returns the number of bytes written.
52	**fs.writeSync(fd, data[, position[, encoding]])**	Synchronous versions of fs.write(). Returns the number of bytes written.
53	**fs.read(fd, buffer, offset, length, position, callback)**	Read data from the file specified by fd.
54	**fs.readSync(fd, buffer, offset, length, position)**	Synchronous version of fs.read. Returns the number of bytesRead.
55	**fs.readFile(filename[, options], callback)**	Asynchronously reads the entire contents of a file.
56	**fs.readFileSync(filename[, options])**	Synchronous version of fs.readFile. Returns the contents of the filename.
57	**fs.writeFile(filename, data[, options], callback)**	Asynchronously writes data to a file, replacing the file if it already exists. data can be a string or a buffer.
58	**fs.writeFileSync(filename, data[, options])**	The synchronous version of fs.writeFile.
59	**fs.appendFile(filename, data[, options], callback)**	Asynchronously append data to a file, creating the file if it not yet exists. data can be a string or a buffer.
60	**fs.appendFileSync(filename, data[, options])**	The synchronous version of fs.appendFile.
61	**fs.watchFile(filename[, options], listener)**	Watch for changes on filename. The callback listener will be called each time the file is accessed.
62	**fs.unwatchFile(filename[, listener])**	Stop watching for changes on filename. If listener is specified, only that particular listener is removed. Otherwise, all listeners are removed and you have effectively stopped watching filename.
63	**fs.watch(filename[, options][, listener])**	Watch for changes on filename, where filename is either a file or a directory. The returned object is a fs.FSWatcher.
64	**fs.exists(path, callback)**	Test whether or not the given path exists by checking with the file system. Then call the callback argument with either true or false.
65	**fs.existsSync(path)**	Synchronous version of fs.exists.

66	**fs.access(path[, mode], callback)**	Tests a user's permissions for the file specified by path. mode is an optional integer that specifies the accessibility checks to be performed.
67	**fs.accessSync(path[, mode])**	Synchronous version of fs.access. This throws if any accessibility checks fail, and does nothing otherwise.
68	**fs.createReadStream(path[, options])**	Returns a new ReadStream object.
69	**fs.createWriteStream(path[, options])**	Returns a new WriteStream object.
70	**fs.symlink(srcpath, dstpath[, type], callback)**	Asynchronous symlink(). No arguments other than a possible exception are given to the completion callback. The type argument can be set to 'dir', 'file', or 'junction' (default is 'file') and is only available on Windows (ignored on other platforms). Note that Windows junction points require the destination path to be absolute. When using 'junction', the destination argument will automatically be normalized to absolute path.

Flags

flags for read/write operations are:

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. Instructs the operating system to bypass the local file system cache. This is primarily useful for opening files on NFS mounts as it allows you to skip the potentially stale local cache. It has a very real impact on I/O performance so don't use this flag unless you need it. Note that this doesn't turn fs.open() into a synchronous blocking call. If that's what you want then you should be using fs.openSync()

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

Create a txt file named test.txt in **C:\>Nodejs_WorkSpace**

```
TutorialsPoint.Com
```

Create a js file named test.js in **C:\>Nodejs_WorkSpace**.

File: test.js

```
var fs = require("fs");
var buffer = new Buffer(1024);

//Example: Opening File
function openFile(){
   console.log("\nOpen file");
   fs.open('test.txt', 'r+', function(err,fd) {
      if (err) console.log(err.stack);
          console.log("File opened");
   });
}

//Example: Getting File Info
function getStats(){
   console.log("\nGetting File Info");
   fs.stat('test.txt', function (err, stats) {
      if (err) console.log(err.stack);
      console.log(stats);
      console.log("isFile ? "+stats.isFile());
      console.log("isDirectory ? "+stats.isDirectory());
   });
}

//Example: Writing File
function writeFile(){
   console.log("\nWrite file");
   fs.open('test1.txt', 'w+', function(err,fd) {
      var data = "TutorialsPoint.com - Simply Easy Learning!";
      buffer.write(data);

      fs.write(fd, buffer,0,data.length,0,function(err, bytes){
         if (err) console.log(err.stack);
             console.log(bytes + " written!");
      });
   });
}

//Example: Read File
function readFile(){
   console.log("\nRead file");
   fs.open('test1.txt', 'r+', function(err,fd) {
      if (err) console.log(err.stack);
      fs.read(fd, buffer,0,buffer.length,0,function(err, bytes){
         if (err) console.log(err.stack);
             console.log(bytes + " read!");
             if(bytes > 0){
            console.log(buffer.slice(0,bytes).toString());
             }
      });
   });
}

function closeFile(){
   console.log("\nClose file");
   fs.open('test.txt', 'r+', function(err,fd) {
      if (err) console.log(err.stack);
          fs.close(fd,function(){
             if (err) console.log(err.stack);
                 console.log("File closed!");
          });
   });
```

```
}

function deleteFile(){
   console.log("\nDelete file");
   fs.open('test1.txt', 'r+', function(err,fd) {
           fs.unlink('test1.txt', function(err) {
                  if (err) console.log(err.stack);
                  console.log("File deleted!");
           });
   });
}

function truncateFile(){
   console.log("\nTruncate file");
   fs.open('test.txt', 'r+', function(err,fd) {
           fs.ftruncate(fd, function(err) {
                  if (err) console.log(err.stack);
                  console.log("File truncated!");
           });
   });
}

function createDirectory(){
        console.log("\nCreate Directory");
        fs.mkdir('test',function(err){
      if(!err){
             console.log("Directory created!");
          }
      if(err && err.code === 'EEXIST'){
         console.log("Directory exists!");
      } else if (err) {
         console.log(err.stack);
      }
        });
}

function removeDirectory(){
        console.log("\nRemove Directory");
        fs.rmdir('test',function(err){
      if(!err){
             console.log("Directory removed!");
          }
      if (err) {
                 console.log("Directory do not exist!");
      }
        });
}

function watchFile(){
   fs.watch('test.txt', function (event, filename) {
      console.log('event is: ' + event);
   });
}

//Opening file
openFile();

//Writing File
writeFile();

//Reading File
```

```
readFile();

//Closing Files
closeFile();

//Getting file information
getStats();

//Deleting Files
deleteFile();

//Truncating Files
truncateFile();

//Creating Directories
createDirectory();

//Removing Directories
removeDirectory();

//Watching File Changes
watchFile();
```

Now run the test.js to see the result:

```
C:\Nodejs_WorkSpace>node test.js
```

Verify the Output.

```
Open file

Write file

Read file

Close file

Getting File Info

Delete file

Truncate file

Create Directory

Remove Directory
File opened
{ dev: 0,
  mode: 33206,
  nlink: 1,
```

```
  uid: 0,
  gid: 0,
  rdev: 0,
  ino: 0,
  size: 0,
  atime: Fri Jan 01 2010 00:02:15 GMT+0530 (India Standard Time),
  mtime: Sun Feb 15 2015 13:33:09 GMT+0530 (India Standard Time),
  ctime: Fri Jan 01 2010 00:02:15 GMT+0530 (India Standard Time) }
isFile ? true
isDirectory ? false
Directory created!
Directory removed!
event is: rename
event is: rename
42 written!
42 read!
TutorialsPoint.com - Simply Easy Learning!
File closed!
File deleted!
File truncated!
event is: change
```

Node.js - Utility Modules

In this article, we'll discuss some of the utility modules provided by Node.js library which are very common and are frequently used across the applications.

Sr.No.	Module Name & Description
1	**Console** Used to print information on stdout and stderr.
2	**Process** Used to get information on current process. Provides multiple events related to process activities.
3	**OS Module** Provides basic operating-system related utility functions.
4	**Path Module** Provides utilities for handling and transforming file paths.
5	**Net Module** Provides both servers and clients as streams. Acts as a network wrapper.
6	**DNS Module** Provides functions to do actual DNS lookup as well as to use underlying

	operating system name resolution functionalities.
7	**Domain Module** Provides way to handle multiple different I/O operations as a single group.

Node.js - Console

console is a global object and is used to print to stdout and stderr. It is used in synchronous way when destination is file or a terminal and asynchronous way when destination is a pipe.

Methods

Sr. No.	method	Description
1	**console.log([data][, ...])**	Prints to stdout with newline. This function can take multiple arguments in a printf()-like way.
2	**console.info([data][, ...])**	Prints to stdout with newline. This function can take multiple arguments in a printf()-like way.
3	**console.error([data][, ...])**	Prints to stderr with newline. This function can take multiple arguments in a printf()-like way.
4	**console.warn([data][, ...])**	Prints to stderr with newline. This function can take multiple arguments in a printf()-like way
5	**console.dir(obj[, options])**	Uses util.inspect on obj and prints resulting string to stdout.
6	**console.time(label)**	Mark a time.
7	**console.timeEnd(label)**	Finish timer, record output.
8	**console.trace(message[, ...])**	Print to stderr 'Trace :', followed by the formatted message and stack trace to the current position.
9	**console.assert(value[, message][, ...])**	Similar to assert.ok(), but the error message is formatted as util.format(message...).

Example

Create a js file named test.js in **C:\>Nodejs_WorkSpace**.

File: test.js

```
var counter = 10;

console.log("Counter: %d", counter);

console.time("Getting data");
//make a database call to retrive the data
//getDataFromDataBase();
console.timeEnd('Getting data');

console.info("Program Ended!")
```

Now run the test.js to see the result:

```
C:\Nodejs_WorkSpace>node test.js
```

Verify the Output.

```
Counter: 10
Getting data: 0ms
Program Ended!
```

Node.js - Process

process is a global object and is used to represent Node process.

Exit Codes

Node normally exit with a 0 status code when no more async operations are pending. There are other exit codes which are described below:

Code	Name	Description
1	**Uncaught Fatal Exception**	There was an uncaught exception, and it was not handled by a domain or an uncaughtException event handler.
2	**Unused**	reserved by Bash for builtin misuse
3	**Internal JavaScript Parse Error**	The JavaScript source code internal in Node's bootstrapping process caused a parse error. This is extremely rare, and generally can only happen during development of Node itself.
4	**Internal JavaScript Evaluation Failure**	The JavaScript source code internal in Node's bootstrapping process failed to return a function value when evaluated. This is extremely rare, and generally can only happen during development of Node itself.
5	**Fatal Error**	There was a fatal unrecoverable error in V8. Typically a message

		will be printed to stderr with the prefix FATAL ERROR.
6	**Non-function Internal Exception Handler**	There was an uncaught exception, but the internal fatal exception handler function was somehow set to a non-function, and could not be called.
7	**Internal Exception Handler Run-Time Failure**	here was an uncaught exception, and the internal fatal exception handler function itself threw an error while attempting to handle it.
8	**Unused**	
9	**Invalid Argument**	Either an unknown option was specified, or an option requiring a value was provided without a value.
10	**Internal JavaScript Run-Time Failure**	The JavaScript source code internal in Node's bootstrapping process threw an error when the bootstrapping function was called. This is extremely rare, and generally can only happen during development of Node itself.
12	**Invalid Debug Argument**	The --debug and/or --debug-brk options were set, but an invalid port number was chosen.
>128	**Signal Exits**	If Node receives a fatal signal such as SIGKILL or SIGHUP, then its exit code will be 128 plus the value of the signal code. This is a standard Unix practice, since exit codes are defined to be 7-bit integers, and signal exits set the high-order bit, and then contain the value of the signal code.

Events

Process is an eventEmitter and it emits the following events.

Sr.No.	Event	Description
1	**exit**	Emitted when the process is about to exit. There is no way to prevent the exiting of the event loop at this point, and once all exit listeners have finished running the process will exit.
2	**beforeExit**	This event is emitted when node empties it's event loop and has nothing else to schedule. Normally, node exits when there is no work scheduled, but a listener for

		'beforeExit' can make asynchronous calls, and cause node to continue.
3	**uncaughtException**	Emitted when an exception bubbles all the way back to the event loop. If a listener is added for this exception, the default action (which is to print a stack trace and exit) will not occur.
4	**Signal Events**	Emitted when the processes receives a signal such as SIGINT, SIGHUP, etc.

Properties

Process provides many useful properties to get better control over system interactions.

Sr.No.	Property	Description
1	**stdout**	A Writable Stream to stdout.
2	**stderr**	A Writable Stream to stderr.
3	**stdin**	A Writable Stream to stdin.
4	**argv**	An array containing the command line arguments. The first element will be 'node', the second element will be the name of the JavaScript file. The next elements will be any additional command line arguments.
5	**execPath**	This is the absolute pathname of the executable that started the process.
6	**execArgv**	This is the set of node-specific command line options from the executable that started the process.
7	**env**	An object containing the user environment.
8	**exitCode**	A number which will be the process exit code, when the process either exits gracefully, or is exited via process.exit() without specifying a code.
9	**version**	A compiled-in property that exposes NODE_VERSION.
10	**versions**	A property exposing version strings of node and its dependencies.
11	**config**	An Object containing the JavaScript representation of the configure options that were used to compile the current node executable. This is the same as the "config.gypi" file that was produced when running the ./configure script.

12	**pid**	The PID of the process.
13	**title**	Getter/setter to set what is displayed in 'ps'.
14	**arch**	What processor architecture you're running on: 'arm', 'ia32', or 'x64'.
15	**platform**	What platform you're running on: 'darwin', 'freebsd', 'linux', 'sunos' or 'win32'
16	**mainModule**	Alternate way to retrieve require.main. The difference is that if the main module changes at runtime, require.main might still refer to the original main module in modules that were required before the change occurred. Generally it's safe to assume that the two refer to the same module.

Methods

Process provides many useful methods to get better control over system interactions.

Sr.No.	Method	Description
1	**abort()**	This causes node to emit an abort. This will cause node to exit and generate a core file.
2	**chdir(directory)**	Changes the current working directory of the process or throws an exception if that fails.
3	**cwd()**	Returns the current working directory of the process.
4	**exit([code])**	Ends the process with the specified code. If omitted, exit uses the 'success' code 0.
5	**getgid()**	Gets the group identity of the process. This is the numerical group id, not the group name.This function is only available on POSIX platforms (i.e. not Windows, Android).
6	**setgid(id)**	Sets the group identity of the process. (See setgid(2).) This accepts either a numerical ID or a groupname string. If a groupname is specified, this method blocks while resolving it to a numerical ID.This function is only available on POSIX platforms (i.e. not Windows, Android).
7	**getuid()**	Gets the user identity of the process. This is the numerical id, not the username.This function is only available on POSIX platforms (i.e. not Windows,

		Android).
8	**setuid(id)**	Sets the user identity of the process. (See setgid(2).) This accepts either a numerical ID or a username string. If a username is specified, this method blocks while resolving it to a numerical ID.This function is only available on POSIX platforms (i.e. not Windows, Android).
9	**getgroups()**	Returns an array with the supplementary group IDs. POSIX leaves it unspecified if the effective group ID is included but node.js ensures it always is.This function is only available on POSIX platforms (i.e. not Windows, Android).
10	**setgroups(groups)**	Sets the supplementary group IDs. This is a privileged operation, meaning you need to be root or have the CAP_SETGID capability.This function is only available on POSIX platforms (i.e. not Windows, Android).
11	**initgroups(user, extra_group)**	Reads /etc/group and initializes the group access list, using all groups of which the user is a member. This is a privileged operation, meaning you need to be root or have the CAP_SETGID capability.This function is only available on POSIX platforms (i.e. not Windows, Android).
12	**kill(pid[, signal])**	Send a signal to a process. pid is the process id and signal is the string describing the signal to send. Signal names are strings like 'SIGINT' or 'SIGHUP'. If omitted, the signal will be 'SIGTERM'.
13	**memoryUsage()**	Returns an object describing the memory usage of the Node process measured in bytes.
14	**nextTick(callback)**	Once the current event loop turn runs to completion, call the callback function.
15	**umask([mask])**	Sets or reads the process's file mode creation mask. Child processes inherit the mask from the parent process. Returns the old mask if mask argument is given, otherwise returns the current mask.
16	**uptime()**	Number of seconds Node has been running.
17	**hrtime()**	Returns the current high-resolution real time in a [seconds, nanoseconds] tuple Array. It is relative to an arbitrary time in the past. It is not related to the time of day and therefore not subject to clock drift. The

		primary use is for measuring performance between intervals.

Example

Create a js file named test.js in **C:\>Nodejs_WorkSpace**.

File: test.js

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

//printing to console
process.stdout.write("Hello World!" + "\n");

//reading passed parameter
process.argv.forEach(function(val, index, array) {
   console.log(index + ': ' + val);
});

//executable path
console.log(process.execPath);

//print the current directory
console.log('Current directory: ' + process.cwd());

//print the process version
console.log('Current version: ' + process.version);

//print the memory usage
console.log(util.inspect(process.memoryUsage()));
```

Now run the test.js to see the result:

```
C:\Nodejs_WorkSpace>node test.js
```

Verify the Output.

```
Hello World!
0: node
1: C:\Nodejs_WorkSpace\test.js
2: one
3: 2
4: three
C:\Program Files\nodejs\node.exe
Current directory: C:\Nodejs_WorkSpace
Current version: v0.10.36
{ rss: 9314304, heapTotal: 3047296, heapUsed: 1460196 }
```

Node.js - OS Module

os module is used for few basic operating-system related utility functions. os

module can be imported using following syntax.

```
var os = require("os")
```

Methods

Sr. No.	method	Description
1	**os.tmpdir()**	Returns the operating system's default directory for temp files.
2	**os.endianness()**	Returns the endianness of the CPU. Possible values are "BE" or "LE".
3	**os.hostname()**	Returns the hostname of the operating system.
4	**os.type()**	Returns the operating system name.
5	**os.platform()**	Returns the operating system platform.
6	**os.arch()**	Returns the operating system CPU architecture. Possible values are "x64", "arm" and "ia32".
7	**os.release()**	Returns the operating system release.
8	**os.uptime()**	Returns the system uptime in seconds.
9	**os.loadavg()**	Returns an array containing the 1, 5, and 15 minute load averages.
10	**os.totalmem()**	Returns the total amount of system memory in bytes.
11	**os.freemem()**	Returns the amount of free system memory in bytes.
12	**os.cpus()**	Returns an array of objects containing information about each CPU/core installed: model, speed (in MHz), and times (an object containing the number of milliseconds the CPU/core spent in: user, nice, sys, idle, and irq).
13	**os.networkInterfaces()**	Get a list of network interfaces.

Properties

Sr. No.	property	Description
1	**os.EOL**	A constant defining the appropriate End-of-line marker for the operating system.

Example

Create a js file named test.js in **C:\>Nodejs_WorkSpace**.

File: test.js

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

//endianness
console.log('endianness : ' + os.endianness());

//type
console.log('type : ' + os.type());

//platform
console.log('platform : ' + os.platform());

//totalmem
console.log('total memory : ' + os.totalmem() + " bytes.");

//freemem
console.log('free memory : ' + os.freemem() + " bytes.");
```

Now run the test.js to see the result:

```
C:\Nodejs_WorkSpace>node test.js
```

Verify the Output.

```
endianness : LE
type : Windows_NT
platform : win32
total memory : 1072152576 bytes.
free memory : 461508608 bytes.
```

Node.js - Path Module

path module is used for handling and transforming file paths. path module can be imported using following syntax.

```
var path = require("path")
```

Properties

Process provides many useful properties to get better control over system interactions.

Sr.No.	Property	Description
1	**path.sep**	

		The platform-specific file separator. '\\' or '/'.
2	**path.delimiter**	The platform-specific path delimiter, ; or ':'.
3	**path.posix**	Provide access to aforementioned path methods but always interact in a posix compatible way.
4	**path.win32**	Provide access to aforementioned path methods but always interact in a win32 compatible way.

Methods

Sr. No.	method	Description
1	**path.normalize(p)**	Normalize a string path, taking care of '..' and '.' parts.
2	**path.join([path1][, path2][, ...])**	Join all arguments together and normalize the resulting path.
3	**path.resolve([from ...], to)**	Resolves to to an absolute path.
4	**path.isAbsolute(path)**	Determines whether path is an absolute path. An absolute path will always resolve to the same location, regardless of the working directory.
5	**path.relative(from, to)**	Solve the relative path from from to to.
6	**path.dirname(p)**	Return the directory name of a path. Similar to the Unix dirname command.
7	**path.basename(p[, ext])**	Return the last portion of a path. Similar to the Unix basename command.
8	**path.extname(p)**	Return the extension of the path, from the last '.' to end of string in the last portion of the path. If there is no '.' in the last portion of the path or the first character of it is '.', then it returns an empty string.
9	**path.parse(pathString)**	Returns an object from a path string.
10	**path.format(pathObject)**	Returns a path string from an object, the opposite of path.parse above.

Example

Create a js file named test.js in **C:\>Nodejs_WorkSpace**.

File: test.js

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

//normalization
console.log('normalization : ' + path.normalize('/test/test1//2slashes/1slash/tab/..'));

//join
console.log('joint path : ' + path.join('/test', 'test1', '2slashes/1slash', 'tab', '..'));

//resolve
console.log('resolve : ' + path.resolve('test.js'));

//extName
console.log('ext name : ' + path.extname('test.js'));
```

Now run the test.js to see the result:

```
C:\Nodejs_WorkSpace>node test.js
```

Verify the Output.

```
normalization : \test\test1\2slashes\1slash
joint path : \test\test1\2slashes\1slash
resolve : C:\Nodejs_WorkSpace\test.js
ext name : .js
```

Node.js - Net Module

net module is used to create both servers and clients. It provides an aynchronous network wrapper. net module can be imported using following syntax.

```
var net = require("net")
```

Methods

Sr. No.	method	Description
1	**net.createServer([options][, connectionListener])**	Creates a new TCP server. The connectionListener argument is automatically set as a listener for the 'connection' event.
2	**net.connect(options[, connectionListener])**	A factory method, which returns a new 'net.Socket' and connects to the supplied address and port.
3	**net.createConnection(options[, connectionListener])**	A factory method, which returns a new 'net.Socket' and connects to the supplied address and port.

4	**net.connect(port[, host][, connectListener])**	Creates a TCP connection to port on host. If host is omitted, 'localhost' will be assumed. The connectListener parameter will be added as an listener for the 'connect' event. Is a factory method which returns a new 'net.Socket'.
5	**net.createConnection(port[, host][, connectListener])**	Creates a TCP connection to port on host. If host is omitted, 'localhost' will be assumed. The connectListener parameter will be added as an listener for the 'connect' event. Is a factory method which returns a new 'net.Socket'.
6	**net.connect(path[, connectListener])**	Creates unix socket connection to path. The connectListener parameter will be added as an listener for the 'connect' event. A factory method which returns a new 'net.Socket'.
7	**net.createConnection(path[, connectListener])**	Creates unix socket connection to path. The connectListener parameter will be added as an listener for the 'connect' event. A factory method which returns a new 'net.Socket'.
8	**net.isIP(input)**	Tests if input is an IP address. Returns 0 for invalid strings, returns 4 for IP version 4 addresses, and returns 6 for IP version 6 addresses.
9	**net.isIPv4(input)**	Returns true if input is a version 4 IP address, otherwise returns false.
10	**net.isIPv6(input)**	Returns true if input is a version 6 IP address, otherwise returns false.

Class:net.Server

This class is used to create a TCP or local server.

Methods

Sr. No.	method	Description
1	**server.listen(port[, host][, backlog][, callback])**	Begin accepting connections on the specified port and host. If the host is omitted, the server will accept connections directed to any IPv4 address (INADDR_ANY). A port

		value of zero will assign a random port.
2	**server.listen(path[, callback])**	Start a local socket server listening for connections on the given path.
3	**server.listen(handle[, callback])**	The handle object can be set to either a server or socket (anything with an underlying _handle member), or a {fd: <n>} object. This will cause the server to accept connections on the specified handle, but it is presumed that the file descriptor or handle has already been bound to a port or domain socket. Listening on a file descriptor is not supported on Windows.
4	**server.listen(options[, callback])**	The port, host, and backlog properties of options, as well as the optional callback function, behave as they do on a call to server.listen(port, [host], [backlog], [callback]) . Alternatively, the path option can be used to specify a UNIX socket.
5	**server.close([callback])**	finally closed when all connections are ended and the server emits a 'close' event.
6	**server.address()**	Returns the bound address, the address family name and port of the server as reported by the operating system.
7	**server.unref()**	Calling unref on a server will allow the program to exit if this is the only active server in the event system. If the server is already unrefd calling unref again will have no effect.
8	**server.ref()**	Opposite of unref, calling ref on a previously unrefd server will not let the program exit if it's the only server left (the default behavior). If the server is refd calling ref again will have no effect.
9	**server.getConnections(callback)**	Asynchronously get the number of concurrent connections on the server. Works when sockets were sent to forks. Callback should take two arguments err and count.

Events

Sr.		

No.	event	Description
1	**listening**	Emitted when the server has been bound after calling server.listen.
2	**connection**	Emitted when a new connection is made. Socket object, The connection object is available to event handler. Socket is an instance of net.Socket.
3	**close**	Emitted when the server closes. Note that if connections exist, this event is not emitted until all connections are ended.
4	**error**	Emitted when an error occurs. The 'close' event will be called directly following this event.

Class:net.Socket

This object is an abstraction of a TCP or local socket. net.Socket instances implement a duplex Stream interface. They can be created by the user and used as a client (with connect()) or they can be created by Node and passed to the user through the 'connection' event of a server.

Events

net.Socket is an eventEmitter and it emits the following events.

Sr.No.	Event	Description
1	**lookup**	Emitted after resolving the hostname but before connecting. Not applicable to UNIX sockets.
2	**connect**	Emitted when a socket connection is successfully established.
3	**data**	Emitted when data is received. The argument data will be a Buffer or String. Encoding of data is set by socket.setEncoding().
4	**end**	Emitted when the other end of the socket sends a FIN packet.
5	**timeout**	Emitted if the socket times out from inactivity. This is only to notify that the socket has been idle. The user must manually close the connection.
6	**drain**	Emitted when the write buffer becomes empty. Can be used to throttle uploads.
7	**error**	Emitted when an error occurs. The 'close' event will be called directly following this event.
8	**close**	Emitted once the socket is fully closed. The argument had_error is a boolean which says if the socket was closed due to a transmission

		error.

Properties

net.Socket provides many useful properties to get better control over socket interactions.

Sr.No.	Property	Description
1	**socket.bufferSize**	This property shows the number of characters currently buffered to be written.
2	**socket.remoteAddress**	The string representation of the remote IP address. For example, '74.125.127.100' or '2001:4860:a005::68'.
3	**socket.remoteFamily**	The string representation of the remote IP family. 'IPv4' or 'IPv6'.
4	**socket.remotePort**	The numeric representation of the remote port. For example, 80 or 21.
5	**socket.localAddress**	The string representation of the local IP address the remote client is connecting on. For example, if you are listening on '0.0.0.0' and the client connects on '192.168.1.1', the value would be '192.168.1.1'.
6	**socket.localPort**	The numeric representation of the local port. For example, 80 or 21.
7	**socket.bytesRead**	The amount of received bytes.
8	**socket.bytesWritten**	The amount of bytes sent.

Methods

Sr. No.	method	Description
1	**new net.Socket([options])**	Construct a new socket object.
2	**socket.connect(port[, host][, connectListener])**	Opens the connection for a given socket. If port and host are given, then the socket will be opened as a TCP socket, if host is omitted, localhost will be assumed. If a path is given, the socket will be opened as a unix socket to that path.
		Opens the connection for a given socket. If port and host are given, then the socket will

3	**socket.connect(path[, connectListener])**	be opened as a TCP socket, if host is omitted, localhost will be assumed. If a path is given, the socket will be opened as a unix socket to that path.
4	**socket.setEncoding([encoding])**	Set the encoding for the socket as a Readable Stream.
5	**socket.write(data[, encoding][, callback])**	Sends data on the socket. The second parameter specifies the encoding in the case of a string--it defaults to UTF8 encoding.
6	**socket.end([data][, encoding])**	Half-closes the socket. i.e., it sends a FIN packet. It is possible the server will still send some data.
7	**socket.destroy()**	Ensures that no more I/O activity happens on this socket. Only necessary in case of errors (parse error or so).
8	**socket.pause()**	Pauses the reading of data. That is, 'data' events will not be emitted. Useful to throttle back an upload.
9	**socket.resume()**	Resumes reading after a call to pause().
10	**socket.setTimeout(timeout[, callback])**	Sets the socket to timeout after timeout milliseconds of inactivity on the socket. By default net.Socket do not have a timeout.
11	**socket.setNoDelay([noDelay])**	Disables the Nagle algorithm. By default TCP connections use the Nagle algorithm, they buffer data before sending it off. Setting true for noDelay will immediately fire off data each time socket.write() is called. noDelay defaults to true.
12	**socket.setKeepAlive([enable][, initialDelay])**	Enable/disable keep-alive functionality, and optionally set the initial delay before the first keepalive probe is sent on an idle socket. enable defaults to false.
13	**socket.address()**	Returns the bound address, the address family name and port of the socket as reported by the operating system. Returns an object with three properties, e.g. { port: 12346, family: 'IPv4', address: '127.0.0.1' }.

14	**socket.unref()**	Calling unref on a socket will allow the program to exit if this is the only active socket in the event system. If the socket is already unrefd calling unref again will have no effect.
15	**socket.ref()**	Opposite of unref, calling ref on a previously unrefd socket will not let the program exit if it's the only socket left (the default behavior). If the socket is refd calling ref again will have no effect.

Example

Create a js file named server.js in **C:\>Nodejs_WorkSpace**.

File: server.js

```
var net = require('net');
var server = net.createServer(function(connection) {
   console.log('client connected');
   connection.on('end', function() {
      console.log('client disconnected');
   });
   connection.write('Hello World!\r\n');
   connection.pipe(connection);
});
server.listen(8080, function() {
  console.log('server is listening');
});
```

Now run the server.js to see the result:

```
C:\Nodejs_WorkSpace>node server.js
```

Verify the Output.

```
server is listening
```

Create a js file named client.js in **C:\>Nodejs_WorkSpace**.

File: client.js

```
var net = require('net');
var client = net.connect({port: 8080}, function() {
   console.log('connected to server!');
});
client.on('data', function(data) {
   console.log(data.toString());
   client.end();
});
client.on('end', function() {
   console.log('disconnected from server');
});
```

Now run the client.js in another terminal to see the result:

```
C:\Nodejs_WorkSpace>node client.js
```

Verify the Output.

```
connected to server!
Hello World!


disconnected from server
```

Verify the Output on terminal where server.js is running.

```
server is listening
client connected
client disconnected
```

Node.js - DNS Module

dns module is used to do actual DNS lookup as well as to use underlying operating system name resolution functionalities.. It provides an aynchronous network wrapper. dns module can be imported using following syntax.

```
var dns = require("dns")
```

Methods

Sr. No.	method	Description
1	**dns.lookup(hostname[, options], callback)**	Resolves a hostname (e.g. 'google.com') into the first found A (IPv4) or AAAA (IPv6) record. options can be an object or integer. If options is not provided, then IP v4 and v6 addresses are both valid. If options is an integer, then it must be 4 or 6.

2	**dns.lookupService(address, port, callback)**	Resolves the given address and port into a hostname and service using getnameinfo.
3	**dns.resolve(hostname[, rrtype], callback)**	Resolves a hostname (e.g. 'google.com') into an array of the record types specified by rrtype.
4	**dns.resolve4(hostname, callback)**	The same as dns.resolve(), but only for IPv4 queries (A records). addresses is an array of IPv4 addresses (e.g. ['74.125.79.104', '74.125.79.105', '74.125.79.106']).
5	**dns.resolve6(hostname, callback)**	The same as dns.resolve4() except for IPv6 queries (an AAAA query).
6	**dns.resolveMx(hostname, callback)**	The same as dns.resolve(), but only for mail exchange queries (MX records).
7	**dns.resolveTxt(hostname, callback)**	The same as dns.resolve(), but only for text queries (TXT records). addresses is an 2-d array of the text records available for hostname (e.g., [['v=spf1 ip4:0.0.0.0 ', '~all']]). Each sub-array contains TXT chunks of one record. Depending on the use case, the could be either joined together or treated separately.
8	**dns.resolveSrv(hostname, callback)**	The same as dns.resolve(), but only for service records (SRV records). addresses is an array of the SRV records available for hostname. Properties of SRV records are priority, weight, port, and name (e.g., [{'priority': 10, 'weight': 5, 'port': 21223, 'name': 'service.example.com'}, ...]).
9	**dns.resolveSoa(hostname, callback)**	The same as dns.resolve(), but only for start of authority record queries (SOA record).
10	**dns.resolveNs(hostname, callback)**	The same as dns.resolve(), but only for name server records (NS records). addresses is an array of the name server records available for hostname (e.g., ['ns1.example.com', 'ns2.example.com']).
11	**dns.resolveCname(hostname, callback)**	The same as dns.resolve(), but only for canonical name records (CNAME records). addresses is an array of the canonical name records available for hostname (e.g., ['bar.example.com']).
12	**dns.reverse(ip, callback)**	Reverse resolves an ip address to an array of

		hostnames.
13	**dns.getServers()**	Returns an array of IP addresses as strings that are currently being used for resolution.
14	**dns.setServers(servers)**	Given an array of IP addresses as strings, set them as the servers to use for resolving.

rrtypes

Following is the list of valid rrtypes used by dns.resolve() method

A - IPV4 addresses, default

AAAA - IPV6 addresses

MX - mail exchange records

TXT - text records

SRV - SRV records

PTR - used for reverse IP lookups

NS - name server records

CNAME - canonical name records

SOA - start of authority record

Error Codes

Each DNS query can return one of the following error codes:

dns.NODATA - DNS server returned answer with no data.

dns.FORMERR - DNS server claims query was misformatted.

dns.SERVFAIL - DNS server returned general failure.

dns.NOTFOUND - Domain name not found.

dns.NOTIMP - DNS server does not implement requested operation.

dns.REFUSED - DNS server refused query.

dns.BADQUERY - Misformatted DNS query.

dns.BADNAME - Misformatted hostname.

dns.BADFAMILY - Unsupported address family.

dns.BADRESP - Misformatted DNS reply.

dns.CONNREFUSED - Could not contact DNS servers.

dns.TIMEOUT - Timeout while contacting DNS servers.

dns.EOF - End of file.

dns.FILE - Error reading file.

dns.NOMEM - Out of memory.

dns.DESTRUCTION - Channel is being destroyed.

dns.BADSTR - Misformatted string.

dns.BADFLAGS - Illegal flags specified.

dns.NONAME - Given hostname is not numeric.

dns.BADHINTS - Illegal hints flags specified.

dns.NOTINITIALIZED - c-ares library initialization not yet performed.

dns.LOADIPHLPAPI - Error loading iphlpapi.dll.

dns.ADDRGETNETWORKPARAMS - Could not find GetNetworkParams function.

dns.CANCELLED - DNS query cancelled.

Example

Create a js file named test.js in **C:\>Nodejs_WorkSpace**.

File: test.js

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

dns.lookup('www.google.com', function onLookup(err, address, family) {
   console.log('address:', address);
   dns.reverse(address, function (err, hostnames) {
   if (err) {
      console.log(err.stack);
   }

   console.log('reverse for ' + address + ': ' + JSON.stringify(hostnames));
});
});
```

Now run the test.js to see the result:

```
C:\Nodejs_WorkSpace>node test.js
```

Verify the Output.

```
address: 74.125.200.103
reverse for 74.125.200.103: ["sa-in-f103.1e100.net"]
```

Node.js - Domain Module

domain module is used to intercept unhandled error. These unhandled error can be intercepted using internal binding or external binding. If errors are not handled at all then they will simply crash the Node application.

> **Internal Binding** - Error emmitter is executing its code within run method of a domain.
>
> **External Binding** - Error emmitter is added explicitly to a domain using its add method.

domain module can be imported using following syntax.

```
var domain = require("domain")
```

Domain class of domain module is used to provide functionality of routing errors and uncaught exceptions to the active Domain object. It is a child class of EventEmitter. To handle the errors that it catches, listen to its error event. It is created using following syntax:

```
var domain = require("domain");
var domain1 = domain.create();
```

Methods

Sr.	method	Description

No.		
1	**domain.run(function)**	Run the supplied function in the context of the domain, implicitly binding all event emitters, timers, and lowlevel requests that are created in that context.This is the most basic way to use a domain.
2	**domain.add(emitter)**	Explicitly adds an emitter to the domain. If any event handlers called by the emitter throw an error, or if the emitter emits an error event, it will be routed to the domain's error event, just like with implicit binding.
3	**domain.remove(emitter)**	The opposite of domain.add(emitter). Removes domain handling from the specified emitter.
4	**domain.bind(callback)**	The returned function will be a wrapper around the supplied callback function. When the returned function is called, any errors that are thrown will be routed to the domain's error event.
5	**domain.intercept(callback)**	This method is almost identical to domain.bind(callback). However, in addition to catching thrown errors, it will also intercept Error objects sent as the first argument to the function.
6	**domain.enter()**	The enter method is plumbing used by the run, bind, and intercept methods to set the active domain. It sets domain.active and process.domain to the domain, and implicitly pushes the domain onto the domain stack managed by the domain module (see domain.exit() for details on the domain stack). The call to enter delimits the beginning of a chain of asynchronous calls and I/O operations bound to a domain.
7	**domain.exit()**	The exit method exits the current domain, popping it off the domain stack. Any time execution is going to switch to the context of a different chain of asynchronous calls, it's important to ensure that the current domain is exited. The call to exit delimits either the end of or an interruption to the chain of asynchronous calls and I/O operations bound to a domain.
		Once dispose has been called, the domain will no longer be used by callbacks bound into the

8	**domain.dispose()**	domain via run, bind, or intercept, and a dispose event is emit

Properties

Sr.No.	Property	Description
1	**domain.members**	An array of timers and event emitters that have been explicitly added to the domain.

Example

Create a js file named test.js in **C:\>Nodejs_WorkSpace**.

File: test.js

```
var EventEmitter = require("events").EventEmitter;
var domain = require("domain");

var emitter1 = new EventEmitter();

//Create a domain
var domain1 = domain.create();

domain1.on('error', function(err){
   console.log("domain1 handled this error ("+err.message+")");
});

//explicit binding
domain1.add(emitter1);

emitter1.on('error',function(err){
   console.log("listener handled this error ("+err.message+")");
});

emitter1.emit('error',new Error('To be handled by listener'));

emitter1.removeAllListeners('error');

emitter1.emit('error',new Error('To be handled by domain1'));

var domain2 = domain.create();

domain2.on('error', function(err){
   console.log("domain2 handled this error ("+err.message+")");
});

//implicit binding
domain2.run(function(){
   var emitter2 = new EventEmitter();

   emitter2.emit('error',new Error('To be handled by domain2'));
});


domain1.remove(emitter1);

emitter1.emit('error',new Error('Converted to exception. System will crash!'));
```

Now run the test.js to see the result:

```
C:\Nodejs_WorkSpace>node test.js
```

Verify the Output.

```
listener handled this error (To be handled by listener)
domain1 handled this error (To be handled by domain1)
domain2 handled this error (To be handled by domain2)


events.js:72
        throw er; // Unhandled 'error' event
              ^
```

```
Error: Converted to exception. System will crash!
    at Object.<anonymous> (C:\Nodejs_WorkSpace\test.js:42:23)
    at Module._compile (module.js:456:26)
    at Object.Module._extensions..js (module.js:474:10)
    at Module.load (module.js:356:32)
    at Function.Module._load (module.js:312:12)
    at Function.Module.runMain (module.js:497:10)
    at startup (node.js:119:16)
    at node.js:929:3
```

Node.js - Web Module

Introduction to Web Server

Web Server is a software application which processes request using HTTP protocol and returns web pages as response to the clients. Web servers usually delivers html documents along with images, style sheets and scripts. Most web server also 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. Web server then returns the output of the application server to client.

Apache web server is one of the most common web server being used. It is an open source project.

Path locating process

Web server maps the path of a file using URL, Uniform Resource Locator. It can be a local file system or a external/internal program. For example:

A client makes a request using browser, URL: http://www.test-example-site.com/website/index.htm.

Browser will make request as:

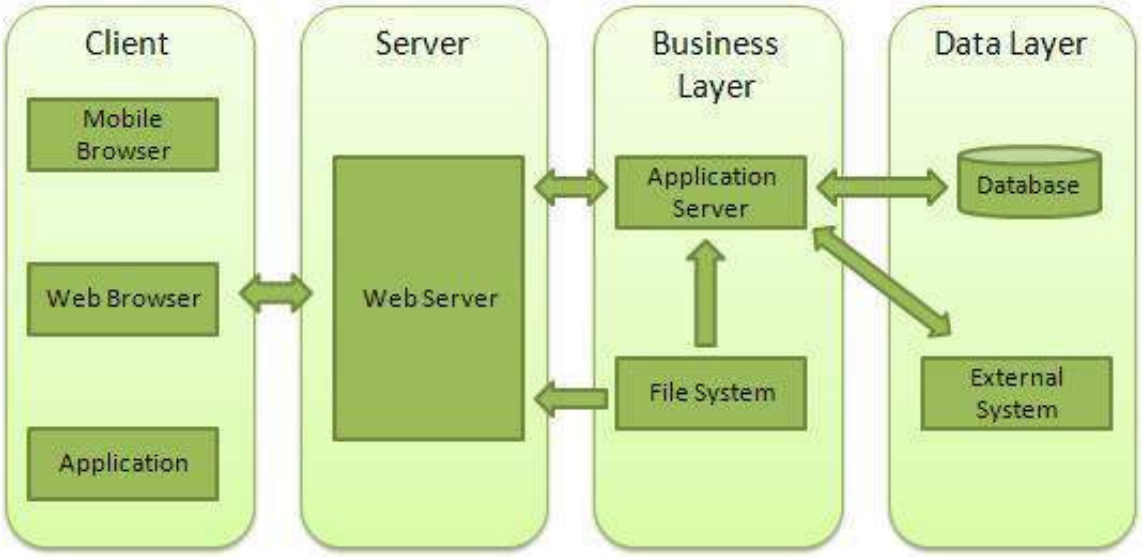
GET /website/index.htm HTTP /1.1

HOST www.test-example-site.com

Web Server will append the path to its root directory. Consider,for example the root directory is home/www then actual path will be translated to home/www/website/index.htm.

Introduction to web architecture

Web application are using divided into four layers:



Client - This layer consists of web browsers, mobile browsers or applications which can make HTTP request to 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 dynamic tasks. 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

Create an HTTP server using http.createServer method. Pass it a function with parameters request and response. Write the sample implementation to return a requested page. Pass a port 8081 to listen method.

Create a js file named server.js in **C:\>Nodejs_WorkSpace**.

File: server.js

```
//http module is required to create a web server
var http = require('http');
//fs module is required to read file from file system
var fs = require('fs');
//url module is required to parse the URL passed to server
var url = require('url');

//create the server
http.createServer(function (request, response) {
   //parse the pathname containing file name
   var pathname = url.parse(request.url).pathname;
   //print the name of the file for which request is made.
   //if url is http://localhost:8081/test.htm then
   //pathname will be /test.htm
   console.log("Request for " + pathname + " received.");
   //read the requested file content from file system
   fs.readFile(pathname.substr(1), function (err, data) {
      //if error occured during file read
      //send a error response to client
      //that web page is not found.
      if (err) {
         console.log(err.stack);
         // 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/');
```

Create a htm file named test.htm in **C:\>Nodejs_WorkSpace**.

File: test.htm

```
<html>
<head>
<title>Sample Page</title>
</head>
<body>
Hello World!
</body>
</html>
```

Now run the server.js to see the result:

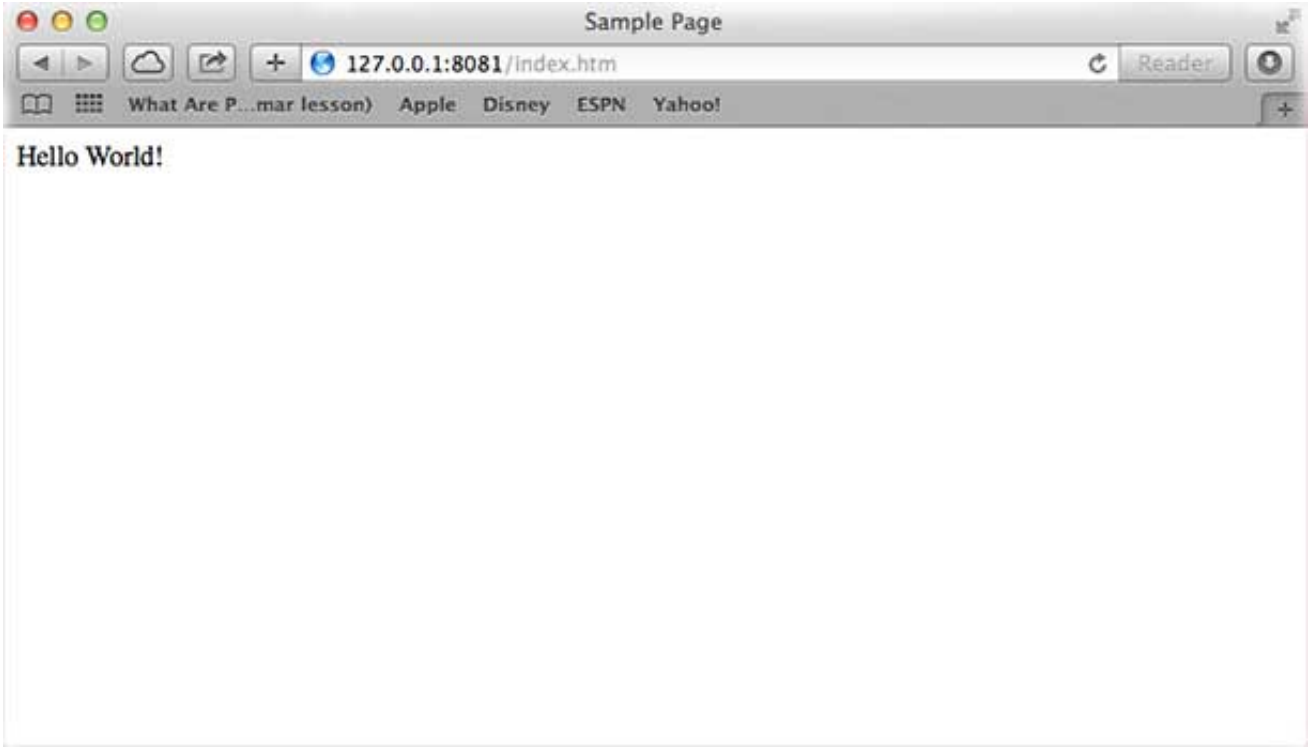
```
C:\Nodejs_WorkSpace>node server.js
```

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/test.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 /test.htm received.
```

Creating Web client using Node

A web client can be created using http module. See the below example:

Create a js file named client.js in **C:\>Nodejs_WorkSpace**.

File: client.js

```
//http module is required to create a web client
var http = require('http');

//options are to be used by request
var options = {
   host: 'localhost',
   port: '8081',
   path: '/test.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 in a different command terminal other than of server.js to see the result:

```
C:\Nodejs_WorkSpace>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 /test.htm received.
Request for /test.htm received.
```

Node.js - Express Application

Express Overview

Express JS is a very popular web application framework built to create Node JS Web based applications. It provides an integrated environment to facilitate rapid development of Node based Web applications. Express framework is based on Connect middleware engine and used Jade html template framework for HTML templating. 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.

```
C:\Nodejs_WorkSpace>npm install express - g
```

Once npm completes the download, you can verify by looking at the content of **<user-directory>**/npm/node_modules. Or type the following command:

```
C:\Nodejs_WorkSpace>npm ls -g
```

You will see the following output:

```
C:\Documents and Settings\Administrator\Application Data\npm
+-- express@4.11.2
  +-- accepts@1.2.3
  | +-- mime-types@2.0.8
  | | +-- mime-db@1.6.1
  | +-- negotiator@0.5.0
  +-- content-disposition@0.5.0
  +-- cookie@0.1.2
  +-- cookie-signature@1.0.5
  +-- debug@2.1.1
  | +-- ms@0.6.2
  +-- depd@1.0.0
  +-- escape-html@1.0.1
  +-- etag@1.5.1
  | +-- crc@3.2.1
  +-- finalhandler@0.3.3
  +-- fresh@0.2.4
```

```
+-- media-typer@0.3.0
+-- merge-descriptors@0.0.2
+-- methods@1.1.1
+-- on-finished@2.2.0
| +-- ee-first@1.1.0
+-- parseurl@1.3.0
+-- path-to-regexp@0.1.3
+-- proxy-addr@1.0.6
| +-- forwarded@0.1.0
| +-- ipaddr.js@0.1.8
+-- qs@2.3.3
+-- range-parser@1.0.2
+-- send@0.11.1
| +-- destroy@1.0.3
| +-- mime@1.2.11
| +-- ms@0.7.0
+-- serve-static@1.8.1
+-- type-is@1.5.6
| +-- mime-types@2.0.8
|   +-- mime-db@1.6.1
+-- utils-merge@1.0.0
+-- vary@1.0.0
```

Express Generator

Now install the express generator using npm. Express generator is used to create an application skeleton using express command.

```
C:\Nodejs_WorkSpace> npm install express-generator -g
```

You will see the following output:

```
C:\Nodejs_WorkSpace>npm install express-generator -g
C:\Documents and Settings\Administrator\Application Data\npm\express -> C:\Docum
ents and Settings\Administrator\Application Data\npm\node_modules\express-generator\bin\express
express-generator@4.12.0 C:\Documents and Settings\Administrator\Application Data\npm\node_modu
+-- sorted-object@1.0.0
+-- commander@2.6.0
+-- mkdirp@0.5.0 (minimist@0.0.8)
```

Hello world Example

Now create a sample application say firstApplication using the following command:

```
C:\Nodejs_WorkSpace> express firstApplication
```

You will see the following output:

```
   create : firstApplication
   create : firstApplication/package.json
   create : firstApplication/app.js
   create : firstApplication/public
   create : firstApplication/public/javascripts
   create : firstApplication/public/images
   create : firstApplication/public/stylesheets
   create : firstApplication/public/stylesheets/style.css
   create : firstApplication/routes
   create : firstApplication/routes/index.js
   create : firstApplication/routes/users.js
   create : firstApplication/views
   create : firstApplication/views/index.jade
   create : firstApplication/views/layout.jade
   create : firstApplication/views/error.jade
   create : firstApplication/bin
   create : firstApplication/bin/www

   install dependencies:
     $ cd firstApplication && npm install

   run the app:
     $ DEBUG=firstApplication:* ./bin/www
```

Move to firstApplication folder and install dependencies of firstApplication using the following command:

```
C:\Nodejs_WorkSpace\firstApplication> npm install
```

You will see the following output:

```
debug@2.1.2 node_modules\debug
+-- ms@0.7.0

cookie-parser@1.3.4 node_modules\cookie-parser
+-- cookie-signature@1.0.6
+-- cookie@0.1.2

morgan@1.5.1 node_modules\morgan
+-- basic-auth@1.0.0
+-- depd@1.0.0
+-- on-finished@2.2.0 (ee-first@1.1.0)
```

```
serve-favicon@2.2.0 node_modules\serve-favicon
+-- ms@0.7.0
+-- fresh@0.2.4
+-- parseurl@1.3.0
+-- etag@1.5.1 (crc@3.2.1)

jade@1.9.2 node_modules\jade
+-- character-parser@1.2.1
+-- void-elements@2.0.1
+-- commander@2.6.0
+-- mkdirp@0.5.0 (minimist@0.0.8)
+-- transformers@2.1.0 (promise@2.0.0, css@1.0.8, uglify-js@2.2.5)
+-- with@4.0.1 (acorn-globals@1.0.2, acorn@0.11.0)
+-- constantinople@3.0.1 (acorn-globals@1.0.2)

express@4.12.2 node_modules\express
+-- merge-descriptors@1.0.0
+-- cookie-signature@1.0.6
+-- methods@1.1.1
+-- cookie@0.1.2
+-- fresh@0.2.4
+-- utils-merge@1.0.0
+-- range-parser@1.0.2
+-- escape-html@1.0.1
+-- parseurl@1.3.0
+-- vary@1.0.0
+-- content-type@1.0.1
+-- finalhandler@0.3.3
+-- serve-static@1.9.1
+-- 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)
+-- proxy-addr@1.0.6 (forwarded@0.1.0, ipaddr.js@0.1.8)
+-- send@0.12.1 (destroy@1.0.3, ms@0.7.0, mime@1.3.4)
+-- accepts@1.2.4 (negotiator@0.5.1, mime-types@2.0.9)
+-- type-is@1.6.0 (media-typer@0.3.0, mime-types@2.0.9)

body-parser@1.12.0 node_modules\body-parser
+-- content-type@1.0.1
+-- bytes@1.0.0
```

```
+-- raw-body@1.3.3
+-- depd@1.0.0
+-- qs@2.3.3
+-- iconv-lite@0.4.7
+-- on-finished@2.2.0 (ee-first@1.1.0)
+-- type-is@1.6.0 (media-typer@0.3.0, mime-types@2.0.9)
```

Here express generator has created a complete application structure which you can verify as firstApplication folder gets created in Nodejs_WorkSpace folder with following folders/files:

```
.
+-- app.js
+-- bin
|   +-- www
+-- package.json
+-- public
|   +-- images
|   +-- javascripts
|   +-- stylesheets
|       +-- style.css
+-- routes
|   +-- index.js
|   +-- users.js
+-- views
    +-- error.jade
    +-- index.jade
    +-- layout.jade
```

package.json Application descriptor file contains dependencies list and other attributes of the application which Node utilizes.

app.js Contains initialization code for server.

bin Used to store the applicaion in production mode.

public Used to store the images, stylesheets and javascript files

routes Contains route handlers

views Contains html templates to generate various views for web application.

First Application

app.js is the core engine of express based application. Let's update the default app.js to include port information and creates a server using it. Add the following lines to app.js

```
//set the server port
app.set('port', process.env.PORT || 3000);

//create the server
http.createServer(app).listen(app.get('port'), function(){
  console.log('Express server listening on port ' + app.get('port'));
});
```

Updated app.js

Following are the full content of the app.js file

Update app.js file present in **C:\>Nodejs_WorkSpace\firstApplication**.

File: app.js

```
var express = require('express');
var path = require('path');
var favicon = require('serve-favicon');
var logger = require('morgan');
var cookieParser = require('cookie-parser');
var bodyParser = require('body-parser');
var http = require('http');

var routes = require('./routes/index');
var users = require('./routes/users');

var app = express();

// view engine setup
app.set('port', process.env.PORT || 3000);
app.set('views', path.join(__dirname, 'views'));
app.set('view engine', 'jade');

// uncomment after placing your favicon in /public
//app.use(favicon(__dirname + '/public/favicon.ico'));
app.use(logger('dev'));
app.use(bodyParser.json());
app.use(bodyParser.urlencoded({ extended: false }));
app.use(cookieParser());
app.use(express.static(path.join(__dirname, 'public')));

app.use('/', routes);
app.use('/users', users);

// catch 404 and forward to error handler
app.use(function(req, res, next) {
   var err = new Error('Not Found');
   err.status = 404;
   next(err);
});

// error handlers

// development error handler
// will print stacktrace
if (app.get('env') === 'development') {
   app.use(function(err, req, res, next) {
      res.status(err.status || 500);
      res.render('error', {
         message: err.message,
         error: err
      });
   });
}

// production error handler
// no stacktraces leaked to user
app.use(function(err, req, res, next) {
   res.status(err.status || 500);
   res.render('error', {
      message: err.message,
      error: {}
   });
});

http.createServer(app).listen(app.get('port'), function(){
```

```
  console.log('Express server listening on port ' + app.get('port'));
});

module.exports = app;
```

Now run the app.js to see the result:

```
C:\Nodejs_WorkSpace\firstApplication>node app
```

Verify the Output. Server has started

```
Express server listening on port 3000
```

Make a request to firstApplication

Open http://localhost:3000/ in any browser and see the below result.

Basic Routing

Following code in app.js binds two route handlers.

```
var routes = require('./routes/index');
var users = require('./routes/users');
...
app.use('/', routes);
app.use('/users', users);
```

routes - routes (index.js), route handler handles all request made to home page via localhost:3000

users - users (users.js), route handler handles all request made to /users via localhost:3000/users

Following is the code of **C:\>Nodejs_WorkSpace\firstApplication\routes\index.js** created by express generator.

```
var express = require('express');
var router = express.Router();

/* GET home page. */
router.get('/', function(req, res, next) {
  res.render('index', { title: 'Express' });
});

module.exports = router;
```

When node server gets a request for home page, express router render the index page using **index.jade** template while passing a parameter **title** with value 'Express'. Following are the contents of **C:\>Nodejs_WorkSpace\firstApplication\views\index.jade** template.

```
extends layout

block content
  h1= title
  p Welcome to #{title}
```

Node.js - Enhancing First Application

Overview

In this article, we'll enhance the express js application created in Express Application chapter to do the following functionalities:

Show list of all the users.

Show details of a particular user.

Add details of new user.

Step 1: Create a JSON based database

Firstly, let's create a sample json based database of users.

Create a json file named user.json in **C:\>Nodejs_WorkSpace\firstApplication**.

File: user.json

```
{
   "user1" : {
      "name" : "mahesh",
```

```
         "password" : "password1",
         "profession" : "teacher"
   },
   "user2" : {
      "name" : "suresh",
         "password" : "password2",
         "profession" : "librarian"
   },
   "user3" : {
      "name" : "ramesh",
         "password" : "password3",
         "profession" : "clerk"
   }
}
```

Step 2: Create Users specific Jade Views

Create a **user** directory in **C:\>Nodejs_WorkSpace\firstApplication\views** directory with the following views.

index.jade - View to show list of all users.

new.jade - View to show a form to add a new user.

profile.jade - View to show detail of an user

Create a index.jade in **C:\>Nodejs_WorkSpace\firstApplication\views\users**.

File: index.jade

```
h1 Users

p
   a(href="/users/new/") Create new user
ul
   - for (var username in users) {
   li
      a(href="/users/" + encodeURIComponent(username))= users[username].name
   - };
```

Create a new.jade in **C:\>Nodejs_WorkSpace\firstApplication\views\users**.

File: index.jade

```
h1 New User

form(method="POST" action="/Users/addUser")
        P
                label(for="name") Name

                input#name(name="name")
        P
                label(for="password") Password
```

```
            input#name(name="password")
        P
            label(for="profession") Profession

            input#name(name="profession")
        P
            input(type="submit", value="Create")
```

Create a profile.jade in **C:\>Nodejs_WorkSpace\firstApplication\views\users**.

File: profile.jade

```
h1 Name: #{user.name}

h2 Profession: #{user.profession}
```

Step 3: Update users route handler, users.js

Update users.js in **C:\>Nodejs_WorkSpace\firstApplication\routes**.

File: users.js

```
var express = require('express');
var router = express.Router();

var users = require('../users.json');
/* GET users listing. */
router.get('/', function(req, res) {
   res.render('users/index', { title: 'Users',users:users });
});

/* Get form to add a new user*/
router.get('/new', function(req, res) {
   res.render('users/new', { title: 'New User'});
});

/* Get detail of a new user */
router.get('/:name', function(req, res, next) {
   var user = users[req.params.name]
   if(user){
      res.render('users/profile', { title: 'User Profile', user:user});
   }else{
      next();
   }
});

/* post the form to add new user */
router.post('/addUser', function(req, res, next) {
   if(users[req.body.name]){
      res.send('Conflict', 409);
   }else{
     users[req.body.name] = req.body;
         res.redirect('/users/');
   }
});
```

```
module.exports = router;
```

Now run the app.js to see the result:

```
C:\Nodejs_WorkSpace\firstApplication>node app
```

Verify the Output. Server has started

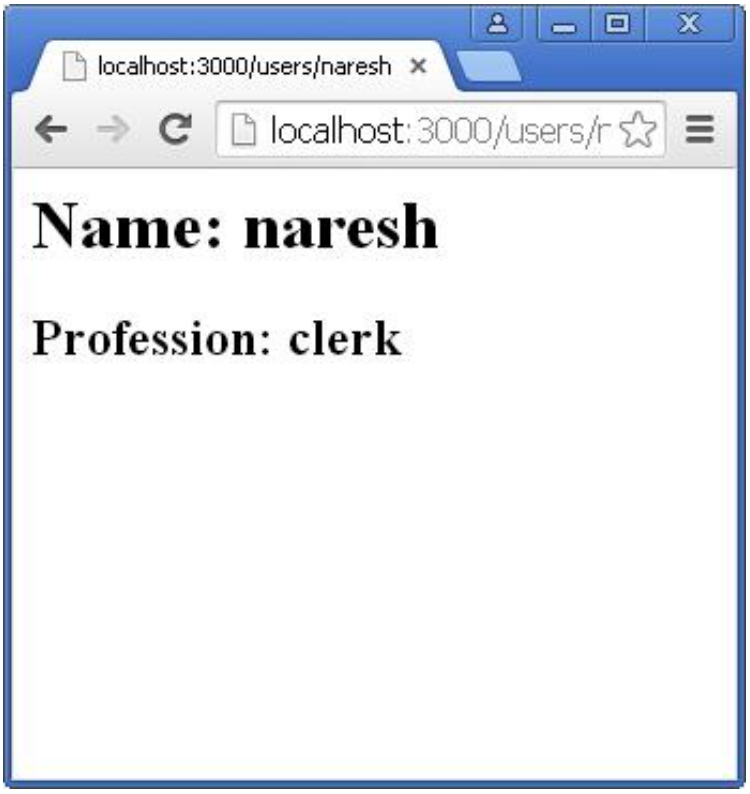
```
Express server listening on port 3000
```

Make a request to firstApplication to get list of all the users. Open http://localhost:3000/users in any browser and see the below result.

Click on Create new User link to see a form.

Submit form and see the updated list.

Click on newly created user to see the details.



You can check the server status also as following:

```
C:\Nodejs_WorkSpace\firstApplication>node app
Express server listening on port 3000
GET /users/ 200 809.161 ms - 201
GET /users/new/ 304 101.627 ms - -
GET /users/new/ 304 33.496 ms - -
POST /Users/addUser 302 56.206 ms - 70
GET /users/ 200 43.548 ms - 245
GET /users/naresh 200 12.313 ms - 47
```

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.

In REST architecture, a REST Server simply provides access to resources and REST client accesses and modifies the resources. Here each resource is identified by URIs/ global IDs. REST uses various representation to represent a resource like text, JSON, XML. JSON is the most popular one.

HTTP methods

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

GET - Provides a read only access to a resource.

PUT - Used to create a new resource.

DELETE - Used to remove a resource.

POST - Used to update a existing resource or create a new resource.

Introduction to 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., 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, provides resource representation such as JSON and set of HTTP Methods.

Creating RESTFul for A Library

INow, we'll enhance the express js application created in Express Sample Application chapter to create a webservice say user management with following

functionalities:

Sr. No.	URI	HTTP Method	POST body	Result
1	**/users/**	**GET**	**empty**	**Show list of all the users.**
2	**/users/addUser**	**POST**	**JSON String**	**Add details of new user.**
3	**/users/:id**	**GET**	**empty**	**Show details of a user.**

Getting All users

Firstly, let's update a sample json based database of users.

Update json file named user.json in **C:\>Nodejs_WorkSpace\firstApplication**.

File: user.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
   }
}
```

When a client send a GET request to /users, server should send a response containing all the user. Update users route handler, users.js

Update users.js in **C:\>Nodejs_WorkSpace\firstApplication\routes**.

File: users.js

```
/* GET users listing. */
router.get('/', function(req, res) {
   res.send({ title: 'Users',users:users });
});
```

Add details of new user.

When a client send a POST request to /users/addUser with body containing the JSON String, server should send a response stating the status. Update users route handler, users.js

Update users.js in **C:\>Nodejs_WorkSpace\firstApplication\routes**.

File: users.js

```
/*add a user*/
router.post('/addUser', function(req, res, next) {
    var body = '';
    req.on('data', function (data) {
        body += data;
    });
    req.on('end', function () {
           var json = JSON.parse(body);
       users["user"+json.id] = body;
           res.send({ Message: 'User Added'});
    });
});
```

Show details of new user.

When a client send a GET request to /users with an id, server should send a response containing detail of that user. Update users route handler, users.js

Update users.js in **C:\>Nodejs_WorkSpace\firstApplication\routes**.

File: users.js

```
router.get('/:id', function(req, res, next) {
   var user = users["user" + req.params.id]
   if(user){
      res.send({ title: 'User Profile', user:user});
   }else{
      res.send({ Message: 'User not present'});
   }
});
```

Complete User.js

File: users.js

```
var express = require('express');
var router = express.Router();

var users = require('../users.json');
/* GET users listing. */
router.get('/', function(req, res) {
```

```
   res.send({ title: 'Users',users:users });
});

router.get('/:id', function(req, res, next) {
   console.log(req.params)
   var user = users["user" + req.params.id]
   if(user){
      res.send({ title: 'User Profile', user:user});
   }else{
      res.send({ Message: 'User not present'});
   }
});

router.post('/addUser', function(req, res, next) {
    var body = '';
    req.on('data', function (data) {
        body += data;
    });
    req.on('end', function () {
           var json = JSON.parse(body);
       users["user"+json.id] = body;
           res.send({ Message: 'User Added'});
    });
});

module.exports = router;
```

Output

We are using Postman , a Chrome extension, to test our webservices.

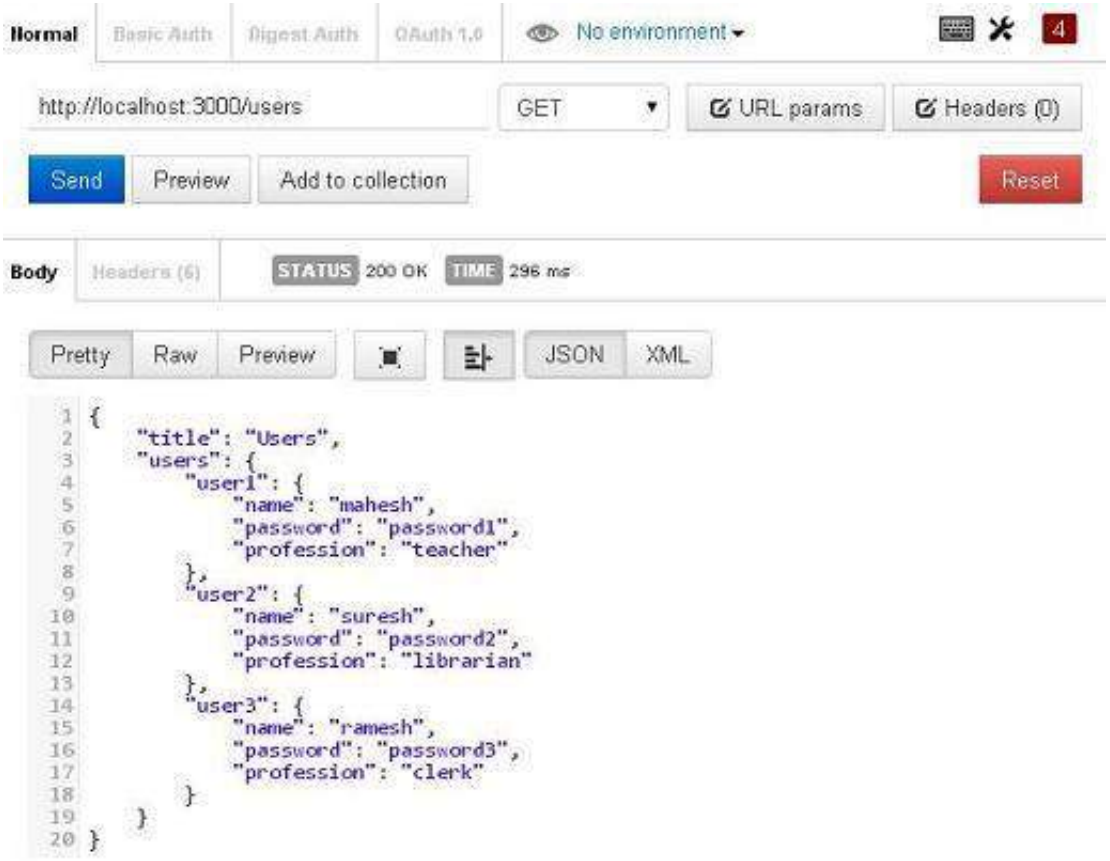
Now run the app.js to see the result:

```
C:\Nodejs_WorkSpace\firstApplication>node app
```

Verify the Output. Server has started

```
Express server listening on port 3000
```

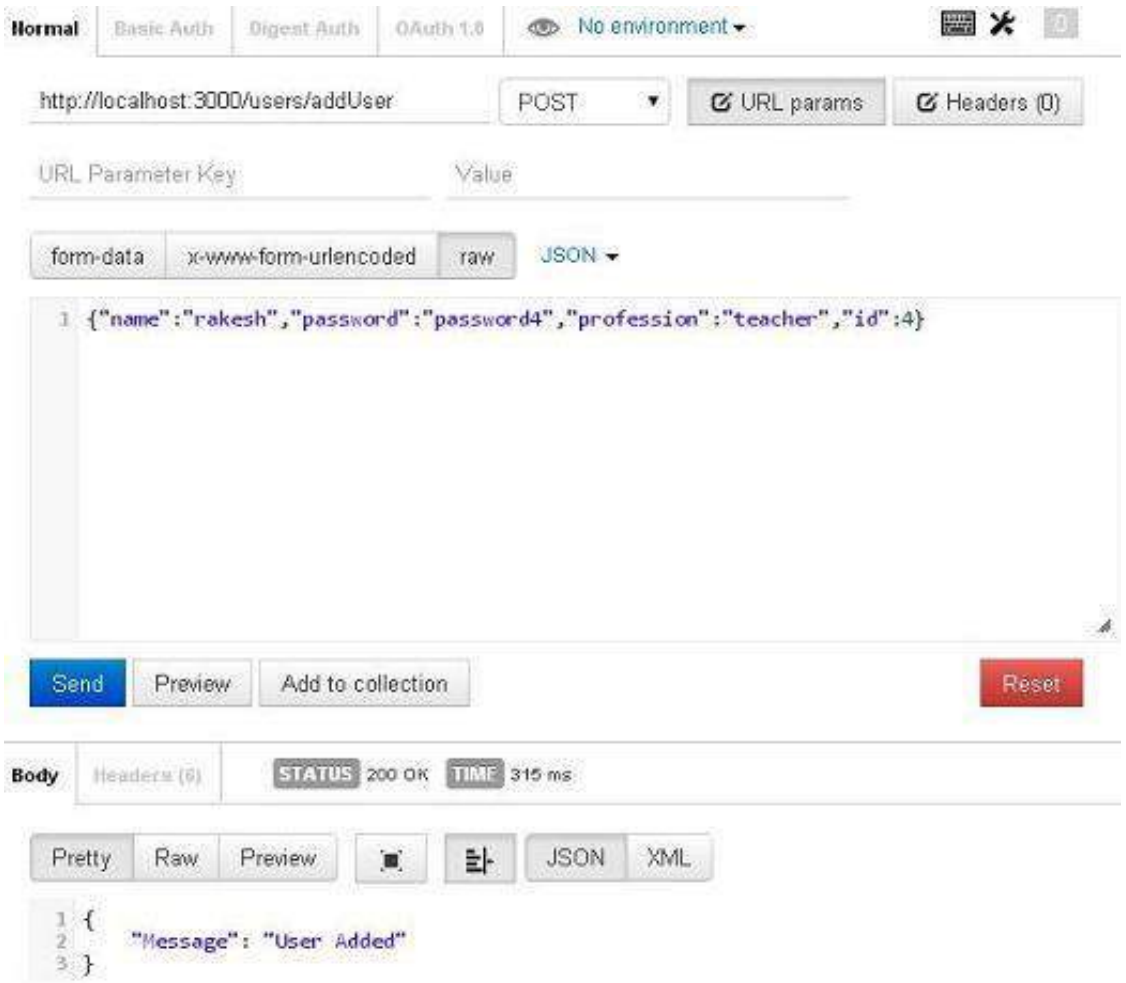
Make a request to firstApplication to get list of all the users. Put http://localhost:3000/users in POSTMAN with GET request and see the below result.



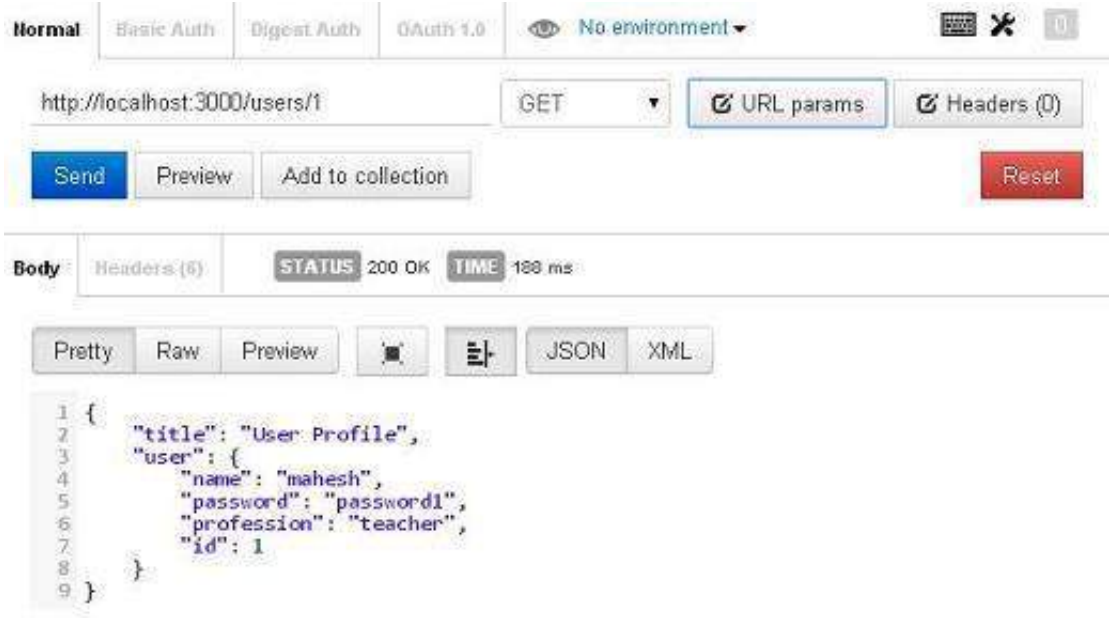
Make a request to firstApplication to add a new user. Put http://localhost:3000/users/addUser in POSTMAN with POST request and see the below result.

Make sure to add json body in and select POST as method.

```
{"name":"rakesh","password":"password4","profession":"teacher","id":4}
```



Make a request to firstApplication to get a user. Put http://localhost:3000/users/1 in POSTMAN with GET request and see the below result.



Node.js - Scaling Application

As node runs in a single thread mode, 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. they may be a separate stream objects which can be piped to and from.

There are 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 Node processes.

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)
```

> **command** String The command to run, with space-separated arguments
>
> **options** Object
>
> > **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 called with the output when process terminates

error Error

stdout Buffer

stderr Buffer

Return: ChildProcess object

exec() 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

Create two js file named worker.js and master.js in **C:\>Nodejs_WorkSpace**.

File: worker.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 worker.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:

```
C:\Nodejs_WorkSpace>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.
```

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])
```

command String The command to run

args Array List of string arguments

options Object

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.

Return: ChildProcess object

spawn() 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 worker.js and master.js in **C:\>Nodejs_WorkSpace**.

File: worker.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', ['worker.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:

```
C:\Nodejs_WorkSpace>node master.js
```

Verify the Output. Server has started

```
stdout: Child Process 0 executed.


child process exited with code 0
stdout: Child Process 2 executed.


child process exited with code 0
stdout: Child Process 1 executed.


child process exited with code 0
```

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])
```

modulePath String The module to run in the child

args Array List of string arguments

options Object

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.

Return: ChildProcess object

fork 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 worker.js and master.js in **C:\>Nodejs_WorkSpace**.

File: worker.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("worker.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:

```
C:\Nodejs_WorkSpace>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
```

Previous Page Next Page

Enter email for newsletter go