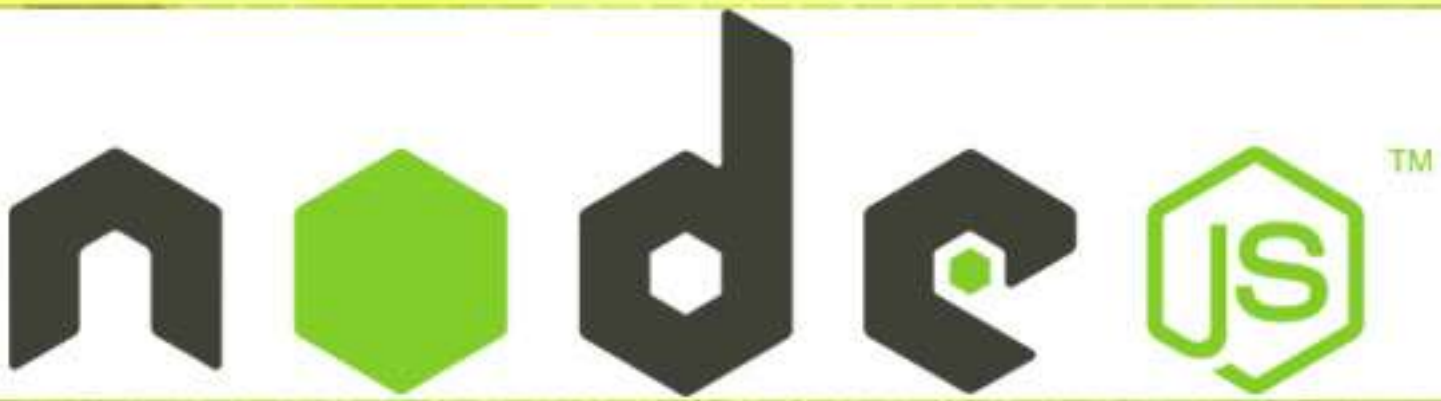


tutorialspoint

SIMPLYEASYLEARNING

www.tutorialspoint.com

https://www.facebook.com/tutorialspointindia

https://twitter.com/tutorialspoint

About the Tutorial

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

Audience

This tutorial is designed for software programmers who want to learn the basics of Node.js and its architectural concepts. This tutorial will give you enough understanding on all the necessary components of Node.js with suitable examples.

Prerequisites

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

Execute Node.js Online

For most of the examples given in this tutorial, you will find a **Try it** option, so just make use of this option to execute your Node.js programs on the spot and enjoy your learning.

Try the following example using the Try it option available at the top right corner of the below sample code box (on our website):

```
/* Hello World! program in Node.js */

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

Copyright & Disclaimer

Table of Contents

1. Introduction

What is Node.js?

Node.js is a server-side platform built on Google Chrome's JavaScript Engine (V8 Engine). Node.js was developed by Ryan Dahl in 2009 and its latest version is v0.10.36. The definition of Node.js as supplied by its official documentation is as follows:

> Node.js is a platform built on Chrome's JavaScript runtime for easily building fast and scalable network applications. Node.js uses an event-driven, non-blocking I/O model that makes it lightweight and efficient, perfect for data-intensive real-time applications that run across distributed devices.

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

Node.js also provides a rich library of various JavaScript modules which simplifies the development of web applications using Node.js to a great extent.

```
Node.js = Runtime Environment + JavaScript Library
```

Features of Node.js

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

- **Asynchronous and Event Driven** – All APIs of Node.js library are asynchronous, that is, non-blocking. It essentially means a Node.js based server never waits for an API to return data. The server moves to the next API after calling it and a notification mechanism of Events of Node.js helps the server to get a response from the previous API call.

- **Very Fast** – Being built on Google Chrome's V8 JavaScript Engine, Node.js library is very fast in code execution.

- **Single Threaded but Highly Scalable** – Node.js uses a single threaded model with event looping. Event mechanism helps the server to respond in a non-blocking way and makes the server highly scalable as opposed to traditional servers which create limited threads to handle requests. Node.js uses a single threaded program and the same program can provide service to a much larger number of requests than traditional servers like Apache HTTP Server.

- **No Buffering** – Node.js applications never buffer any data. These applications simply output the data in chunks.

- **License** – Node.js is released under the **MIT license**.

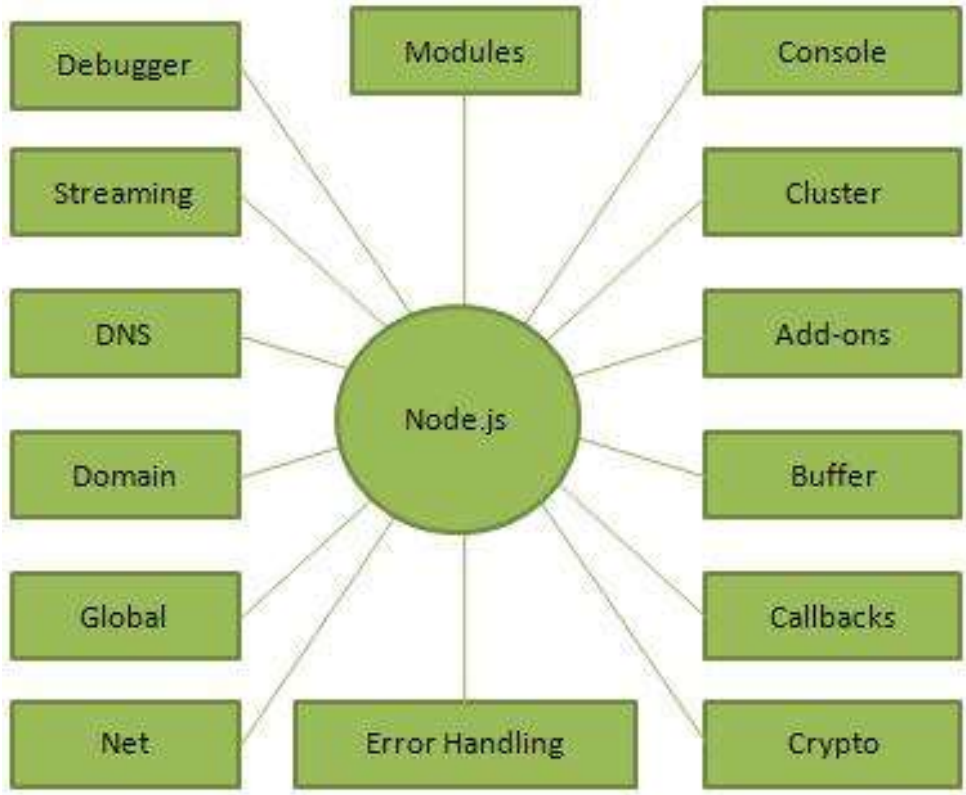
Who Uses Node.js?

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

- **Projects, Applications, and Companies Using Node**

Concepts

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



Where to Use Node.js?

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

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

Where Not to Use Node.js?

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

2. 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 execute all the available examples online and learn through practice. Feel free to modify any example and check the results with different options.

Try the following example using the **Try it** option available at the top right corner of the below sample code box (on our website):

```
/* Hello World! program in Node.js */
console.log("Hello World!");
```

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

Local Environment Setup

If you want to set up your environment for Node.js, you need to have the following two software on your computer, (a) a Text Editor and (b) the Node.js binary installables.

Text Editor

You need to have a text editor to type your program. Examples of text editors include Windows Notepad, OS Edit command, Brief, Epsilon, EMACS, and vim or vi.

The name and version of text editors can vary from one operating system to another. 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 they contain the program source code. The source files for Node.js programs are typically named with the extension ".js".

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

The Node.js Runtime

The source code that you would write in a source file is simply javascript. The Node.js interpreter interprets and executes 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.

The following section explains how to install Node.js binary distribution on various OS.

Download Node.js Archive

Download the latest version of Node.js installable archive file from Node.js Downloads. At the time of writing this tutorial, following are the versions available on different OS.

OS	Archive name
Windows	node-v6.3.1-x64.msi
Linux	node-v6.3.1-linux-x86.tar.gz
Mac	node-v6.3.1-darwin-x86.tar.gz
SunOS	node-v6.3.1-sunos-x86.tar.gz

Installation on UNIX/Linux/Mac OS X and SunOS

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

```
$ cd /tmp
$ wget http://nodejs.org/dist/v6.3.1/node-v6.3.1-linux-x64.tar.gz
$ tar xvfz node-v6.3.1-linux-x64.tar.gz
$ mkdir -p /usr/local/nodejs
$ mv node-v6.3.1-linux-x64/* /usr/local/nodejs
```

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

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

Installation on Windows

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

Verify Installation: Executing a File

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

```
/* Hello, World! program in node.js */
console.log("Hello, World!")
```

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

```
$ node main.js
```

If everything is fine with your installation, it should produce the following result:

```
Hello, World!
```

3. First Application

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

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

2. **Create server**: A server which will listen to client's requests similar to Apache HTTP Server.

3. **Read request and return response**: The server created in an earlier step will read the HTTP request made by the client which can be a browser or a console and return the response.

Creating Node.js Application

Step 1 - Import Required Module

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

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

Step 2 - Create Server

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

```
http.createServer(function (request, response) {

   // Send the HTTP header
   // HTTP Status: 200 : OK
   // Content Type: text/plain
   response.writeHead(200, {'Content-Type': 'text/plain'});

   // Send the response body as "Hello World"
   response.end('Hello World\n');
}).listen(8081);
```

```
// Console will print the message
console.log('Server running at http://127.0.0.1:8081/');
```

The above code is enough to create an HTTP server which listens, i.e., waits for a request over 8081 port on the local machine.

Step 3 - Testing Request & Response

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

```
var http = require("http");
http.createServer(function (request, response) {

   // Send the HTTP header
   // HTTP Status: 200 : OK
   // Content Type: text/plain
   response.writeHead(200, {'Content-Type': 'text/plain'});

   // Send the response body as "Hello World"
   response.end('Hello World\n');
}).listen(8081);

// Console will print the message
console.log('Server running at http://127.0.0.1:8081/');
```

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

```
$ node main.js
```

Verify the Output. Server has started.

```
Server running at http://127.0.0.1:8081/
```

Make a Request to the Node.js Server

Open http://127.0.0.1:8081/ in any browser and observe the following result.

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

4. REPL Terminal

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

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

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

Online REPL Terminal

To simplify your learning, we have set up an easy-to-use Node.js REPL environment online, where you can practice Node.js syntax: Launch Node.js REPL Terminal

Starting REPL

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

```
$ node
```

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

```
$ node
>
```

Simple Expression

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

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

```
>
```

Use Variables

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

```
$ node
> x = 10
10
> var y = 10
undefined
> x + y
20
> console.log("Hello World")
Hello Workd
undefined
```

Multiline Expression

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

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

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

Underscore Variable

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

```
$ node
> var x = 10
undefined
> var y = 20
undefined
> x + y
30
> var sum = _
undefined
> console.log(sum)
30
undefined
>
```

REPL Commands

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

Stopping REPL

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

```
$ node
>
(^C again to quit)
>
```

5. NPM

Node Package Manager (NPM) provides two main functionalities:

- Online repositories for node.js packages/modules which are searchable on search.nodejs.org
- Command line utility to install Node.js packages, do version management and dependency management of Node.js packages.

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

```
$ npm --version
2.7.1
```

If you are running an old version of NPM, then it is quite easy to update it to the latest version. Just use the following command from root:

```
$ sudo npm install npm -g
/usr/bin/npm -> /usr/lib/node_modules/npm/bin/npm-cli.js
npm@2.7.1 /usr/lib/node_modules/npm
```

Installing Modules using NPM

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

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

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

```
$ npm install express
```

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

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

Global vs Local Installation

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

```
$ ls -l
total 0
drwxr-xr-x 3 root root 20 Mar 17 02:23 node_modules
```

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

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

```
$ npm install express -g
```

This will produce a similar result but the module will be installed globally. Here, the first line shows the module version and the location where it is getting installed.

```
express@4.12.2 /usr/lib/node_modules/express
├── merge-descriptors@1.0.0
├── utils-merge@1.0.0
├── cookie-signature@1.0.6
├── methods@1.1.1
├── fresh@0.2.4
├── cookie@0.1.2
├── escape-html@1.0.1
├── range-parser@1.0.2
├── content-type@1.0.1
├── finalhandler@0.3.3
├── vary@1.0.0
├── parseurl@1.3.0
├── content-disposition@0.5.0
├── path-to-regexp@0.1.3
├── depd@1.0.0
├── qs@2.3.3
├── on-finished@2.2.0 (ee-first@1.1.0)
├── etag@1.5.1 (crc@3.2.1)
├── debug@2.1.3 (ms@0.7.0)
├── proxy-addr@1.0.7 (forwarded@0.1.0, ipaddr.js@0.1.9)
├── send@0.12.1 (destroy@1.0.3, ms@0.7.0, mime@1.3.4)
├── serve-static@1.9.2 (send@0.12.2)
├── accepts@1.2.5 (negotiator@0.5.1, mime-types@2.0.10)
└── type-is@1.6.1 (media-typer@0.3.0, mime-types@2.0.10)
```

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

```
$ npm ls -g
```

Using package.json

package.json is present in the root directory 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 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/acceptance/",

   "test-cov": "istanbul cover node_modules/mocha/bin/_mocha -- --
   require test/support/env --reporter dot --check-leaks test/
   test/acceptance/",

   "test-tap": "mocha --require test/support/env --reporter tap –
   check-leaks test/ test/acceptance/",

   "test-travis": "istanbul cover node_modules/mocha/bin/_mocha –
   report lcovonly -- --require test/support/env --reporter spec –
   check-leaks test/ test/acceptance/"
},
"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 the following command to uninstall a Node.js module.

```
$ npm uninstall express
```

Once NPM uninstalls the package, you can verify it by looking at the content of /node_modules/ directory or type the following command:

```
$ npm ls
```

Updating a Module

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

```
$ npm update express
```

Search a Module

Search a package name using NPM.

```
$ npm search express
```

Create a Module

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

```
$ npm init
This utility will walk you through creating a package.json file.
It only covers the most common items, and tries to guess sane defaults.

See 'npm help json' for definitive documentation on these fields
and exactly what they do.

Use 'npm install <pkg> --save' afterwards to install a package and
save it as a dependency in the package.json file.

Press ^C at any time to quit.
name: (webmaster)
```

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

```
$ npm adduser
Username: mcmohd
Password:
Email: (this IS public) mcmohd@gmail.com
```

It is time now to publish your module:

```
$ npm publish
```

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

6. Callback 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 the APIs of Node are written in such a way that they support callbacks.

For example, a function to read a file may start reading a file and return the control to the execution environment immediately so that the 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 a parameter. So there is no blocking or wait for File I/O. This makes Node.js highly scalable, as it can process a high number of requests without waiting for any function to return results.

Blocking Code Example

Create a text file named **input.txt** with the following content:

```
Tutorials Point is giving self learning content
to teach the world in simple and easy way!!!!!
```

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

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

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

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

Now run the main.js to see the result:

```
$ node main.js
```

Verify the Output.

```
Tutorials Point is giving self learning content
to teach the world in simple and easy way!!!!!
Program Ended
```

Non-Blocking Code Example

Create a text file named input.txt with the following content.

```
Tutorials Point is giving self learning content
to teach the world in simple and easy way!!!!!
```

Update main.js to have the following code:

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

fs.readFile('input.txt', function (err, data) {
    if (err) return console.error(err);
    console.log(data.toString());
});

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

Now run the main.js to see the result:

```
$ node main.js
```

Verify the Output.

```
Program Ended
Tutorials Point is giving self learning content
to teach the world in simple and easy way!!!!!
```

These two examples explain the concept of blocking and non-blocking calls.

- The first example shows that the program blocks until it reads the file and then only it proceeds to end the program.

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

Thus, a blocking program executes very much in sequence. From the programming point of view, it is easier to implement the logic but non-blocking programs do not execute in sequence. In case a program needs to use any data to be processed, it should be kept within the same block to make it sequential execution.

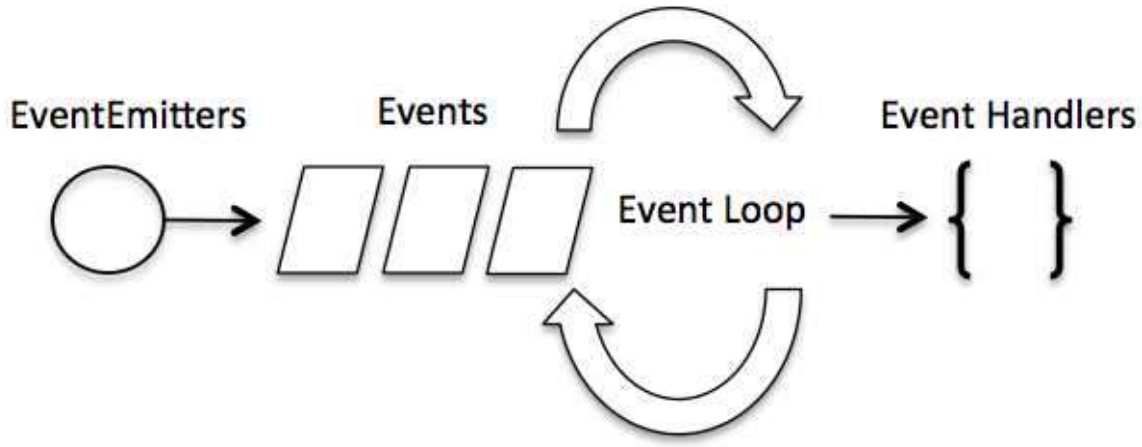
7. Event Loop

Node.js is a single-threaded application, but it can support concurrency via the concept of **event** and **callbacks**. Every API of Node.js is asynchronous and being single-threaded, they use **async function calls** to maintain concurrency. Node uses observer pattern. Node thread keeps an event loop and whenever a task gets completed, it fires the corresponding event which signals the event-listener function to execute.

Event-Driven Programming

Node.js uses events heavily and it is also one of the reasons why Node.js is pretty fast compared to other similar technologies. As soon as Node starts its server, it simply initiates its variables, declares functions, and then simply waits for the event to occur.

In an event-driven application, there is generally a main loop that listens for events, and then triggers a callback function when one of those events is detected.



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

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

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

```
// Bind event and even handler as follows
eventEmitter.on('eventName', eventHandler);
```

We can fire an event programmatically as follows:

```
// Fire an event
eventEmitter.emit('eventName');
```

Example

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

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

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

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

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

// Bind the connection event with the handler
eventEmitter.on('connection', connectHandler);

// Bind the data_received event with the anonymous function
eventEmitter.on('data_received', function(){
   console.log('data received successfully.');
});

// Fire the connection event
eventEmitter.emit('connection');
```

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

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

```
$ mnode main.js
```

It should produce the following result:

```
connection successful.
data received successfully.
Program Ended.
```

How Node Applications Work?

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

```
Tutorials Point is giving self learning content
to teach the world in simple and easy way!!!!!
```

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

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

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

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

```
Program Ended
Tutorials Point is giving self learning content
to teach the world in simple and easy way!!!!!
```

8. Event Emitter

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

EventEmitter Class

As we have seen in the previous section, EventEmitter class lies in the events module. It is accessible via the following code:

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

When an EventEmitter instance faces any error, it emits an 'error' event. When a new listener is added, 'newListener' event is fired and when a listener is removed, 'removeListener' event is fired.

EventEmitter provides multiple properties like **on** and **emit**. **on** property is used to bind a function with the event and **emit** is used to fire an event.

Methods

S.No.	Method & Description
1	**addListener(event, listener)** Adds a listener at 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 at 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 to 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)** Removes a listener from the listener array for the specified event. **Caution**: It changes the 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 the event had listeners, false otherwise.

Class Methods

S.No.	Method & Description
1	**listenerCount(emitter, event)** Returns the number of listeners for a given event.

Events

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

Example

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

```
var events = require('events');
var eventEmitter = new events.EventEmitter();

// listener #1
var listner1 = function listner1() {
   console.log('listner1 executed.');
}

// listener #2
var listner2 = function listner2() {
  console.log('listner2 executed.');
}
```

```
// Bind the connection event with the listner1 function
eventEmitter.addListener('connection', listner1);
// Bind the connection event with the listner2 function
eventEmitter.on('connection', listner2);

var eventListeners =
require('events').EventEmitter.listenerCount(eventEmitter,'connection');
console.log(eventListeners + " Listner(s) listening to connection event");

// Fire the connection event
eventEmitter.emit('connection');

// Remove the binding of listner1 function
eventEmitter.removeListener('connection', listner1);
console.log("Listner1 will not listen now.");

// Fire the connection event
eventEmitter.emit('connection');

eventListeners =
require('events').EventEmitter.listenerCount(eventEmitter,'connection');
console.log(eventListeners + " Listner(s) listening to connection event");

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

Now run the main.js to see the result:

```
$ node main.js
```

Verify the Output.

```
2 Listner(s) listening to connection event
listner1 executed.
listner2 executed.
Listner1 will not listen now.
listner2 executed.
1 Listner(s) listening to connection event
Program Ended.
```

9. Buffers

Pure JavaScript is Unicode friendly, but it is not so for binary data. While dealing with TCP streams or the file system, it's necessary to handle octet streams. Node provides Buffer class which provides instances to store raw data similar to an array of integers but corresponds to a raw memory allocation outside the V8 heap.

Buffer class is a global class that can be accessed in an application without importing the buffer module.

Creating Buffers

Node Buffer can be constructed in a variety of ways.

Method 1

Following is the syntax to create an uninitiated Buffer of 10 octets:

```
var buf = new Buffer(10);
```

Method 2

Following is the syntax to create a Buffer from a given array:

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

Method 3

Following is the syntax to create a Buffer from a given string and optionally encoding type:

```
var buf = new Buffer("Simply Easy Learning", "utf-8");
```

Though "utf8" is the default encoding, you can use any of the following encodings "ascii", "utf8", "utf16le", "ucs2", "base64" or "hex".

Writing to Buffers

Syntax

Following is the syntax of the method to write into a Node Buffer:

```
buf.write(string[, offset][, length][, encoding])
```

Parameters

Here is the description of the parameters used:

- string - This is the string data to be written to buffer.
- offset - This is the index of the buffer to start writing at. Default value is 0.
- length - This is the number of bytes to write. Defaults to buffer.length.
- encoding - Encoding to use. 'utf8' is the default encoding.

Return Value

This method returns the number of octets written. If there is not enough space in the buffer to fit the entire string, it will write a part of the string.

Example

```
buf = new Buffer(256);
len = buf.write("Simply Easy Learning");

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

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

```
Octets written : 20
```

Reading from Buffers

Syntax

Following is the syntax of the method to read data from a Node Buffer:

```
buf.toString([encoding][, start][, end])
```

Parameters

Here is the description of the parameters used:

- encoding - Encoding to use. 'utf8' is the default encoding.
- start - Beginning index to start reading, defaults to 0.
- end - End index to end reading, defaults is complete buffer.

Return Value

This method decodes and returns a string from buffer data encoded using the specified character set encoding.

Example

```
buf = new Buffer(26);
for (var i = 0 ; i < 26 ; i++) {
  buf[i] = i + 97;
}

console.log( buf.toString('ascii'));
// outputs: abcdefghijklmnopqrstuvwxyz
console.log( buf.toString('ascii',0,5));
// outputs: abcde
console.log( buf.toString('utf8',0,5));
// outputs: abcde
console.log( buf.toString(undefined,0,5));
// encoding defaults to 'utf8', outputs abcde
```

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

```
abcdefghijklmnopqrstuvwxyz
abcde
abcde
abcde
```

Convert Buffer to JSON

Syntax

Following is the syntax of the method to convert a Node Buffer into JSON object:

```
buf.toJSON()
```

Return Value

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

Example

```
var buf = new Buffer('Simply Easy Learning');
var json = buf.toJSON(buf);

console.log(json);
```

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

```
[ 83, 105, 109, 112, 108, 121, 32, 69, 97, 115, 121, 32, 76, 101, 97, 114, 110,
105, 110, 103 ]
```

Concatenate Buffers

Syntax

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

```
Buffer.concat(list[, totalLength])
```

Parameters

Here is the description of the parameters used:

- list - Array List of Buffer objects to be concatenated.
- totalLength - This is the total length of the buffers when concatenated.

Return Value

This method returns a Buffer instance.

Example

```
var buffer1 = new Buffer('TutorialsPoint ');
var buffer2 = new Buffer('Simply Easy Learning');
var buffer3 = Buffer.concat([buffer1,buffer2]);
console.log("buffer3 content: " + buffer3.toString());
```

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

```
buffer3 content: TutorialsPoint Simply Easy Learning
```

Compare Buffers

Syntax

Following is the syntax of the method to compare two Node buffers:

```
buf.compare(otherBuffer);
```

Parameters

Here is the description of the parameters used:

- **otherBuffer** - This is the other buffer which will be compared with **buf**.

Return Value

Returns a number indicating whether it comes before or after or is the same as the otherBuffer in sort order.

Example

```
var buffer1 = new Buffer('ABC');
var buffer2 = new Buffer('ABCD');
var result = buffer1.compare(buffer2);

if(result < 0) {
   console.log(buffer1 +" comes before " + buffer2);
}else if(result == 0){
   console.log(buffer1 +" is same as " + buffer2);
}else {
   console.log(buffer1 +" comes after " + buffer2);
}
```

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

```
ABC comes before ABCD
```

Copy Buffer

Syntax

Following is the syntax of the method to copy a node buffer:

```
buf.copy(targetBuffer[, targetStart][, sourceStart][, sourceEnd])
```

Parameters

Here is the description of the parameters used:

- **targetBuffer** - Buffer object where buffer will be copied.
- **targetStart** - Number, Optional, Default: 0
- **sourceStart** - Number, Optional, Default: 0
- **sourceEnd** - Number, Optional, Default: buffer.length

Return Value

No return value. Copies data from a region of this buffer to a region in the target buffer even if the target memory region overlaps with the source. If undefined, the targetStart and sourceStart parameters default to 0, while sourceEnd defaults to buffer.length.

Example

```
var buffer1 = new Buffer('ABC');
//copy a buffer
var buffer2 = new Buffer(3);
buffer1.copy(buffer2);
console.log("buffer2 content: " + buffer2.toString());
```

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

```
buffer2 content: ABC
```

Slice Buffer

Syntax

Following is the syntax of the method to get a sub-buffer of a node buffer:

```
buf.slice([start][, end])
```

Parameters

Here is the description of the parameters used:

- **start** - Number, Optional, Default: 0
- **end** - Number, Optional, Default: buffer.length

Return Value

Returns a new buffer which references the same memory as the old one, but offset and cropped by the start (defaults to 0) and end (defaults to buffer.length) indexes. Negative indexes start from the end of the buffer.

Example

```
var buffer1 = new Buffer('TutorialsPoint');
//slicing a buffer
var buffer2 = buffer1.slice(0,9);
console.log("buffer2 content: " + buffer2.toString());
```

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

```
buffer2 content: Tutorials
```

Buffer Length

Syntax

Following is the syntax of the method to get a size of a node buffer in bytes:

```
buf.length;
```

Return Value

Returns the size of a buffer in bytes.

Example

```
var buffer = new Buffer('TutorialsPoint');
//length of the buffer
console.log("buffer length: " + buffer.length);
```

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

```
buffer length: 14
```

Methods Reference

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

S. No.	Method & Description
1	**new Buffer(size)** Allocates a new buffer of size octets. Note that the size must be no more than kMaxLength. Otherwise, a RangeError will be thrown here.
2	**new Buffer(buffer)** Copies the passed buffer data onto a new Buffer instance.
3	**new Buffer(str[, encoding])** Allocates a new buffer containing the given str. encoding defaults to 'utf8'.
4	**buf.length** Returns the size of the buffer in bytes. Note that this is not necessarily the size of the contents. length refers to the amount of memory allocated for the buffer object. It does not change when the contents of the buffer are changed.

5	**buf.write(string[, offset][, length][, encoding])** Writes a string to the buffer at offset using the given encoding. offset defaults to 0, encoding defaults to 'utf8'. length is the number of bytes to write. Returns the number of octets written.
6	**buf.writeUIntLE(value, offset, byteLength[, noAssert])** Writes a 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.
7	**buf.writeUIntBE(value, offset, byteLength[, noAssert])** Writes a 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.
8	**buf.writeIntLE(value, offset, byteLength[, noAssert])** Writes a 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])** Writes a 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.
10	**buf.readUIntLE(offset, byteLength[, noAssert])** A generalized version of all numeric read methods. Supports up to 48 bits of accuracy. Set noAssert to true to skip validation of offset. It means that the offset may be beyond the end of the buffer. Defaults to false.
11	**buf.readUIntBE(offset, byteLength[, noAssert])** A generalized version of all numeric read methods. Supports up to 48 bits of accuracy. Set noAssert to true to skip validation of offset. It means the offset may be beyond the end of the buffer. Defaults to false.
12	**buf.readIntLE(offset, byteLength[, noAssert])** A generalized version of all numeric read methods. Supports up to 48 bits of accuracy. Set noAssert to true to skip validation of offset. It means the offset may be beyond the end of the buffer. Defaults to false.
13	**buf.readIntBE(offset, byteLength[, noAssert])** A generalized version of all numeric read methods. Supports up to 48 bits of accuracy. Set noAssert to true to skip validation of offset. It means the offset may be beyond the end of the buffer. Defaults to false.
14	**buf.toString([encoding][, start][, end])** 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)** Returns a boolean if this buffer and otherBuffer have the same bytes.
18	**buf.compare(otherBuffer)** Returns a number indicating whether this buffer comes before or after or is the same as the otherBuffer in sort order.
19	**buf.copy(targetBuffer[, targetStart][, sourceStart][, sourceEnd])** 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])** 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])** Reads an unsigned 8-bit integer from the buffer at the specified offset. Set noAssert to true to skip validation of offset. It means the offset may be beyond the end of the buffer. Defaults to false.
22	**buf.readUInt16LE(offset[, noAssert])** Reads an unsigned 16-bit integer from the buffer at the specified offset with the specified endian format. Set noAssert to true to skip validation of offset. It means the offset may be beyond the end of the buffer. Defaults to false.
23	**buf.readUInt16BE(offset[, noAssert])** Reads an unsigned 16-bit integer from the buffer at the specified offset with the specified endian format. Set noAssert to true to skip validation of offset. It means the offset may be beyond the end of the buffer. Defaults to false.
24	**buf.readUInt32LE(offset[, noAssert])** Reads an unsigned 32-bit integer from the buffer at the specified offset with the specified endian format. Set noAssert to true to skip validation of offset. It means the offset may be beyond the end of the buffer. Defaults to false.
25	**buf.readUInt32BE(offset[, noAssert])** Reads an unsigned 32-bit integer from the buffer at the specified offset with the specified endian format. Set noAssert to true to skip validation of offset. It means the offset may be beyond the end of the buffer. Defaults to false.
26	**buf.readInt8(offset[, noAssert])** Reads a signed 8-bit integer from the buffer at the specified offset. Set noAssert to true to skip validation of offset. It means the offset may be beyond the end of the buffer. Defaults to false.
27	**buf.readInt16LE(offset[, noAssert])** Reads a signed 16-bit integer from the buffer at the specified offset with the

	specified endian format. Set noAssert to true to skip validation of offset. It means the offset may be beyond the end of the buffer. Defaults to false.
28	**buf.readInt16BE(offset[, noAssert])** Reads a signed 16-bit integer from the buffer at the specified offset with the specified endian format. Set noAssert to true to skip validation of offset. It means the offset may be beyond the end of the buffer. Defaults to false.
29	**buf.readInt32LE(offset[, noAssert])** Reads a signed 32-bit integer from the buffer at the specified offset with the specified endian format. Set noAssert to true to skip validation of offset. It means the offset may be beyond the end of the buffer. Defaults to false.
30	**buf.readInt32BE(offset[, noAssert])** Reads a signed 32-bit integer from the buffer at the specified offset with the specified endian format. Set noAssert to true to skip validation of offset. It means the offset may be beyond the end of the buffer. Defaults to false.
31	**buf.readFloatLE(offset[, noAssert])** Reads a 32-bit float from the buffer at the specified offset with the specified endian format. Set noAssert to true to skip validation of offset. It means the offset may be beyond the end of the buffer. Defaults to false.
32	**buf.readFloatBE(offset[, noAssert])** Reads a 32-bit float from the buffer at the specified offset with the specified endian format. Set noAssert to true to skip validation of offset. It means the offset may be beyond the end of the buffer. Defaults to false.
33	**buf.readDoubleLE(offset[, noAssert])** Reads a 64-bit double from the buffer at the specified offset with the specified endian format. Set noAssert to true to skip validation of offset. It means the offset may be beyond the end of the buffer. Defaults to false.
34	**buf.readDoubleBE(offset[, noAssert])** Reads a 64-bit double from the buffer at the specified offset with the specified endian format. Set noAssert to true to skip validation of offset. It means the offset may be beyond the end of the buffer. Defaults to false.
35	**buf.writeUInt8(value, offset[, noAssert])** Writes a value to the buffer at the specified offset. Note that the value must be a valid unsigned 8-bit integer. Set noAssert to true to skip validation of value and offset. It means the 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. It should not be used unless you are certain of its correctness. Defaults to false.
36	**buf.writeUInt16LE(value, offset[, noAssert])** Writes a value to the buffer at the specified offset with the specified endian format. Note that the value must be a valid unsigned 16-bit integer. Set noAssert to true to skip validation of value and offset. It means the value may be too large for the specific function and the offset may be beyond the end of the buffer leading to the values being silently dropped. It should not be used unless you are certain of correctness. Defaults to false.
37	**buf.writeUInt16BE(value, offset[, noAssert])** Writes a value to the buffer at the specified offset with the specified endian

	format. Note that the value must be a valid unsigned 16-bit integer. Set noAssert to true to skip validation of value and offset. It means the value may be too large for the specific function and the offset may be beyond the end of the buffer leading to the values being silently dropped. It should not be used unless you are certain of its correctness. Defaults to false.
38	**buf.writeUInt32LE(value, offset[, noAssert])** Writes a value to the buffer at the specified offset with the specified endian format. Note that the value must be a valid unsigned 32-bit integer. Set noAssert to true to skip validation of value and offset. It means the value may be too large for the specific function and the offset may be beyond the end of the buffer leading to the values being silently dropped. It should not be used unless you are certain of its correctness. Defaults to false.
39	**buf.writeUInt32BE(value, offset[, noAssert])** Writes a value to the buffer at the specified offset with the specified endian format. Note that the value must be a valid unsigned 32-bit integer. Set noAssert to true to skip validation of value and offset. It means the value may be too large for the specific function and the offset may be beyond the end of the buffer leading to the values being silently dropped. It should not be used unless you are certain of its correctness. Defaults to false.
40	**buf.writeInt8(value, offset[, noAssert])** Writes a value to the buffer at the specified offset with the specified endian format. Note that the value must be a valid signed 8-bit integer. Set noAssert to true to skip validation of value and offset. It means the value may be too large for the specific function and the offset may be beyond the end of the buffer leading to the values being silently dropped. It should not be used unless you are certain of its correctness. Defaults to false.
41	**buf.writeInt16LE(value, offset[, noAssert])** Writes a value to the buffer at the specified offset with the specified endian format. Note that the value must be a valid signed 16-bit integer. Set noAssert to true to skip validation of value and offset. It means the value may be too large for the specific function and the offset may be beyond the end of the buffer leading to the values being silently dropped. It should not be used unless you are certain of its correctness. Defaults to false.
42	**buf.writeInt16BE(value, offset[, noAssert])** Writes a value to the buffer at the specified offset with the specified endian format. Note that the value must be a valid signed 16-bit integer. Set noAssert to true to skip validation of value and offset. It means the 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. It should not be used unless you are certain of its correctness. Defaults to false.
43	**buf.writeInt32LE(value, offset[, noAssert])** Writes a value to the buffer at the specified offset with the specified endian format. Note that the value must be a valid signed 32-bit integer. Set noAssert to true to skip validation of value and offset. It means the value may be too large for the specific function and the offset may be beyond the end of the buffer leading to the values being silently dropped. It should not be used unless you are certain of its correctness. Defaults to false.
44	**buf.writeInt32BE(value, offset[, noAssert])** Writes a value to the buffer at the specified offset with the specified endian

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

S.No.	Method & Description
1	**Buffer.isEncoding(encoding)** Returns true if the encoding is a valid encoding argument, false otherwise.
2	**Buffer.isBuffer(obj)** Tests if obj is a Buffer.

3	**Buffer.byteLength(string[, encoding])** Gives the actual byte length of a string. encoding defaults to 'utf8'. It is not the same as String.prototype.length, since String.prototype.length returns the number of characters in a string.
4	**Buffer.concat(list[, totalLength])** Returns a buffer which is the result of concatenating all the buffers in the list together.
5	**Buffer.compare(buf1, buf2)** The same as buf1.compare(buf2). Useful for sorting an array of buffers.

10. Streams

What are Streams?

Streams are objects that let you read data from a source or write data to a destination in continuous fashion. In Node.js, there are four types of streams:

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

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

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

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

Reading from a Stream

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

```
Tutorials Point is giving self learning content
to teach the world in simple and easy way!!!!!
```

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

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

// Create a readable stream
var readerStream = fs.createReadStream('input.txt');

// Set the encoding to be utf8.
```

```
readerStream.setEncoding('UTF8');


// Handle stream events --> data, end, and error
readerStream.on('data', function(chunk) {
   data += chunk;
});

readerStream.on('end',function(){
   console.log(data);
});

readerStream.on('error', function(err){
   console.log(err.stack);
});

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

Now run the main.js to see the result:

```
$ node main.js
```

Verify the Output.

```
Program Ended
Tutorials Point is giving self learning content
to teach the world in simple and easy way!!!!!
```

Writing to a Stream

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

```
var fs = require("fs");
var data = 'Simply Easy Learning';

// Create a writable stream
var writerStream = fs.createWriteStream('output.txt');

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

```
// Mark the end of file
writerStream.end();

// Handle stream events --> finish, and error
writerStream.on('finish', function() {
    console.log("Write completed.");
});

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

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

Now run the main.js to see the result:

```
$ node main.js
```

Verify the Output.

```
Program Ended
Write completed.
```

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

```
Simply Easy Learning
```

Piping the Streams

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

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

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

// Create a readable stream
var readerStream = fs.createReadStream('input.txt');

// Create a writable stream
var writerStream = fs.createWriteStream('output.txt');
```

```
// Pipe the read and write operations
// read input.txt and write data to output.txt
readerStream.pipe(writerStream);

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

Now run the main.js to see the result:

```
$ node main.js
```

Verify the Output.

```
Program Ended
```

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

```
Tutorials Point is giving self learning content
to teach the world in simple and easy way!!!!!
```

Chaining the Streams

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

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

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

// Compress the file input.txt to input.txt.gz
fs.createReadStream('input.txt')
  .pipe(zlib.createGzip())
  .pipe(fs.createWriteStream('input.txt.gz'));

console.log("File Compressed.");
```

Now run the main.js to see the result:

```
$ node main.js
```

Verify the Output.

```
File Compressed.
```

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

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

// Decompress the file input.txt.gz to input.txt
fs.createReadStream('input.txt.gz')
  .pipe(zlib.createGunzip())
  .pipe(fs.createWriteStream('input.txt'));

console.log("File Decompressed.");
```

Now run the main.js to see the result:

```
$ node main.js
```

Verify the Output.

```
File Decompressed.
```

11. File System

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

```
ar fs = require("fs")
```

Synchronous vs Asynchronous

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

Example

Create a text file named **input.txt** with the following content:

```
Tutorials Point is giving self learning content
to teach the world in simple and easy way!!!!!
```

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

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

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

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

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

Now run the main.js to see the result:

```
$ node main.js
```

Verify the Output.

```
Synchronous read: Tutorials Point is giving self learning content
to teach the world in simple and easy way!!!!!

Program Ended
Asynchronous read: Tutorials Point is giving self learning content
to teach the world in simple and easy way!!!!!
```

The following sections in this chapter provide a set of good examples on major File I/O methods.

Open a File

Syntax

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

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

Parameters

Here is the description of the parameters used:

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

Flags

Flags for read/write operations are:

Flag	Description
r	Open file for reading. An exception occurs if the file does not exist.
r+	Open file for reading and writing. An exception occurs if the file does not exist.

rs	Open file for reading in synchronous mode.
rs+	Open file for reading and writing, asking 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 the 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 the path exists.
a	Open file for appending. The file is created if it does not exist.
ax	Like 'a' but fails if the path exists.
a+	Open file for reading and appending. The file is created if it does not exist.
ax+	Like 'a+' but fails if the path exists.

Example

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

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

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

Now run the main.js to see the result:

```
$ node main.js
```

Verify the Output.

```
Going to open file!
File opened successfully!
```

Get File Information

Syntax

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

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

Parameters

Here is the description of the parameters used:

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

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

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

Example

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

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

console.log("Going to get file info!");
fs.stat('input.txt', function (err, stats) {
   if (err) {
       return console.error(err);
   }
   console.log(stats);
   console.log("Got file info successfully!");

   // Check file type
   console.log("isFile ? " + stats.isFile());
   console.log("isDirectory ? " + stats.isDirectory());
});
```

Now run the main.js to see the result:

```
$ node main.js
```

Verify the Output.

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

```
isFile ? true
isDirectory ? false
```

Writing a File

Syntax

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

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

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

Parameters

Here is the description of the parameters used:

- **path** - This is the string having the file name including path.
- **data** - This is the String or Buffer to be written into the file.
- **options** - The third parameter is an object which will hold {encoding, mode, flag}. By default, encoding is utf8, mode is octal value 0666, and flag is 'w'
- **callback** - This is the callback function which gets a single parameter err that returns an error in case of any writing error.

Example

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

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

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

```
   });
});
```

Now run the main.js to see the result:

```
$ node main.js
```

Verify the Output.

```
Going to write into existing file
Data written successfully!
Let's read newly written data
Asynchronous read: Simply Easy Learning!
```

Reading a File

Syntax

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

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

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

Parameters

Here is the description of the parameters used:

- **fd** - This is the file descriptor returned by fs.open().
- **buffer** - This is the buffer that the data will be written to.
- **offset** - This is the offset in the buffer to start writing at.
- **length** - This is an integer specifying the number of bytes to read.
- **position** - This is an integer specifying where to begin reading from in the file. If position is null, data will be read from the current file position.
- **callback** - This is the callback function which gets the three arguments, (err, bytesRead, buffer).

Example

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

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

console.log("Going to open an existing file");
fs.open('input.txt', 'r+', function(err, fd) {
   if (err) {
       return console.error(err);
   }
   console.log("File opened successfully!");
   console.log("Going to read the file");
   fs.read(fd, buf, 0, buf.length, 0, function(err, bytes){
      if (err){
         console.log(err);
      }
      console.log(bytes + " bytes read");

      // Print only read bytes to avoid junk.
      if(bytes > 0){
         console.log(buf.slice(0, bytes).toString());
      }
   });
});
```

Now run the main.js to see the result:

```
$ node main.js
```

Verify the Output.

```
Going to open an existing file
File opened successfully!
Going to read the file
97 bytes read
Tutorials Point is giving self learning content
to teach the world in simple and easy way!!!!!
```

Closing a File

Syntax

Following is the syntax to close an opened file:

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

Parameters

Here is the description of the parameters used:

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

Example

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

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

console.log("Going to open an existing file");
fs.open('input.txt', 'r+', function(err, fd) {
   if (err) {
       return console.error(err);
   }
   console.log("File opened successfully!");
   console.log("Going to read the file");
   fs.read(fd, buf, 0, buf.length, 0, function(err, bytes){
      if (err){
         console.log(err);
      }

      // Print only read bytes to avoid junk.
      if(bytes > 0){
         console.log(buf.slice(0, bytes).toString());
      }

      // Close the opened file.
```

```
      fs.close(fd, function(err){
         if (err){
            console.log(err);
         }
         console.log("File closed successfully.");
      });
   });
});
```

Now run the main.js to see the result:

```
$ node main.js
```

Verify the Output.

```
Going to open an existing file
File opened successfully!
Going to read the file
Tutorials Point is giving self learning content
to teach the world in simple and easy way!!!!!

File closed successfully.
```

Truncate a File

Syntax

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

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

Parameters

Here is the description of the parameters used:

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

Example

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

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

console.log("Going to open an existing file");
fs.open('input.txt', 'r+', function(err, fd) {
   if (err) {
       return console.error(err);
   }
   console.log("File opened successfully!");
   console.log("Going to truncate the file after 10 bytes");

   // Truncate the opened file.
   fs.ftruncate(fd, 10, function(err){
      if (err){
         console.log(err);
      }
      console.log("File truncated successfully.");
      console.log("Going to read the same file");
      fs.read(fd, buf, 0, buf.length, 0, function(err, bytes){
         if (err){
            console.log(err);
         }

         // Print only read bytes to avoid junk.
         if(bytes > 0){
            console.log(buf.slice(0, bytes).toString());
         }

         // Close the opened file.
         fs.close(fd, function(err){
            if (err){
               console.log(err);
            }
            console.log("File closed successfully.");
```

```
         });
      });
   });
});
```

Now run the main.js to see the result:

```
$ node main.js
```

Verify the Output.

```
Going to open an existing file
File opened successfully!
Going to truncate the file after 10 bytes
File truncated successfully.
Going to read the same file
Tutorials
File closed successfully.
```

Delete a File

Syntax

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

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

Parameters

Here is the description of the parameters used:

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

Example

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

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


console.log("Going to delete an existing file");
fs.unlink('input.txt', function(err) {
   if (err) {
```

```
        return console.error(err);
    }
    console.log("File deleted successfully!");
});
```

Now run the main.js to see the result:

```
$ node main.js
```

Verify the Output.

```
Going to delete an existing file
File deleted successfully!
```

Create a Directory

Syntax

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

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

Parameters

Here is the description of the parameters used:

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

Example

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

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

console.log("Going to create directory /tmp/test");
fs.mkdir('/tmp/test',function(err){
    if (err) {
        return console.error(err);
    }
    console.log("Directory created successfully!");
```

```
});
```

Now run the main.js to see the result:

```
$ node main.js
```

Verify the Output.

```
Going to create directory /tmp/test
Directory created successfully!
```

Read a Directory

Syntax

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

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

Parameters

Here is the description of the parameters used:

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

Example

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

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

console.log("Going to read directory /tmp");
fs.readdir("/tmp/",function(err, files){
   if (err) {
       return console.error(err);
   }
   files.forEach( function (file){
       console.log( file );
   });
});
```

Now run the main.js to see the result:

```
$ node main.js
```

Verify the Output.

```
Going to read directory /tmp
ccmzx99o.out
ccyCSbkF.out
employee.ser
hsperfdata_apache
test
test.txt
```

Remove a Directory

Syntax

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

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

Parameters

Here is the description of the parameters used:

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

Example

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

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


console.log("Going to delete directory /tmp/test");
fs.rmdir("/tmp/test",function(err){
   if (err) {
       return console.error(err);
   }
   console.log("Going to read directory /tmp");
   fs.readdir("/tmp/",function(err, files){
      if (err) {
          return console.error(err);
```

```
      }
      files.forEach( function (file){
          console.log( file );
      });
   });
});
```

Now run the main.js to see the result:

```
$ node main.js
```

Verify the Output.

```
Going to read directory /tmp
ccmzx99o.out
ccyCSbkF.out
employee.ser
hsperfdata_apache
test.txt
```

Methods Reference

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

S.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 an fs.Stats object.
19	**fs.lstat(path, callback)** Asynchronous lstat(). The callback gets two arguments (err, stats) where stats is an 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 an 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)** Asynchronous 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. No arguments other than a possible exception are given to the completion callback.
48	**fs.fsyncSync(fd)** Synchronous fsync.
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 does not exist. 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 an 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. It 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.

12. Global Objects

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

__filename

The **__filename** represents the filename of the code being executed. This is the resolved absolute path of this code file. For a main program, this is not necessarily the same filename used in the command line. The value inside a module is the path to that module file.

Example

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

```
// Let's try to print the value of __filename


console.log( __filename );
```

Now run the main.js to see the result:

```
$ node main.js
```

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

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

__dirname

The **__dirname** represents the name of the directory that the currently executing script resides in.

Example

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

```
// Let's try to print the value of __dirname


console.log( __dirname );
```

Now run the main.js to see the result:

```
$ node main.js
```

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

```
/web/com/1427091028_21099
```

setTimeout(cb, ms)

The **setTimeout(cb, ms)** global function is used to run callback **cb** after at least **ms** milliseconds. The actual delay depends on external factors like OS timer granularity and system load. A timer cannot span more than 24.8 days.

This function returns an opaque value that represents the timer which can be used to clear the timer.

Example

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

```
function printHello(){
   console.log( "Hello, World!");
}
// Now call above function after 2 seconds
setTimeout(printHello, 2000);
```

Now run the main.js to see the result:

```
$ node main.js
```

Verify the output is printed after a little delay.

```
Hello, World!
```

clearTimeout (t)

The **clearTimeout(t)** global function is used to stop a timer that was previously created with setTimeout(). Here **t** is the timer returned by the setTimeout() function.

Example

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

```
function printHello(){
   console.log( "Hello, World!");
}
// Now call above function after 2 seconds
var t = setTimeout(printHello, 2000);
```

```
// Now clear the timer
clearTimeout(t);
```

Now run the main.js to see the result:

```
$ node main.js
```

Verify the output where you will not find anything printed.

setInterval(cb, ms)

The **setInterval(cb, ms)** global function is used to run callback **cb** repeatedly after at least **ms** milliseconds. The actual delay depends on external factors like OS timer granularity and system load. A timer cannot span more than 24.8 days.

This function returns an opaque value that represents the timer which can be used to clear the timer using the function **clearInterval(t)**.

Example

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

```
function printHello(){
   console.log( "Hello, World!");
}
// Now call above function after 2 seconds
setInterval(printHello, 2000);
```

Now run the main.js to see the result:

```
$ node main.js
```

The above program will execute printHello() after every 2 second. Due to system limitation, this program cannot be executed with Try it option so you can check it in your machine locally.

Global Objects

The following table provides a list of other objects which we use frequently in our applications. For more detail, you can refer to the official documentation.

S.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.

Console Object

Node.js **console** is a global object and is used to print different levels of messages to stdout and stderr. There are built-in methods to be used for printing informational, warning, and error messages.

It is used in synchronous way when the destination is a file or a terminal and in asynchronous way when the destination is a pipe.

Console Methods

Following is a list of methods available with the console global object.

S. 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

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

```
console.info("Program Started");

var counter = 10;
console.log("Counter: %d", counter);

console.time("Getting data");
//
// Do some processing here...
//
console.timeEnd('Getting data');

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

Now run the main.js to see the result:

```
node main.js
```

Verify the Output.

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

Process Object

The process object is a global object and can be accessed from anywhere. There are several methods available in a process object.

Process Events

The process object is an instance of EventEmitter and emits the following events:

S. 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 its event loop and has nothing else to schedule. Normally, the node exits when there is no work scheduled, but a listener for 'beforeExit' can make asynchronous calls, and cause the 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.

Example

Create a js file named main.js with the following code for listening for **exit** event:

```
process.on('exit', function(code) {

  // Following code will never execute.
  setTimeout(function() {
    console.log("This will not run");
  }, 0);

  console.log('About to exit with code:', code);
});
console.log("Program Ended");
```

Now run the main.js to see the result:

```
$ node main.js
```

Verify the Output.

```
Program Ended
About to exit with code: 0
```

Exit Codes

Node normally exits 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 built-in 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 the 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 the 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** There 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 the 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.

Process Properties

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

S. 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 the 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.

Example

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

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

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

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



// Platform Information
console.log(process.platform);
```

Now run the main.js to see the result:

```
$ node main.js
```

Verify the Output while running your program on Linux machine:

```
Hello World!
0: node
1: /web/com/1427106219_25089/main.js
/usr/bin/node
linux
```

Methods Reference

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

S.No.	Method & Description
1	**abort()** It causes the node to emit an abort. It causes the 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 available only on POSIX platforms (i.e. not Windows, Android).
6	**setgid(id)** Sets the group identity of the process (See setgid(2)). It 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 available only 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)). It 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 available only 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 available only on POSIX platforms (i.e. not Windows, Android).
10	**setgroups(groups)** Sets the supplementary group IDs. This is a privileged operation, which implies that you have to be at the root or have the CAP_SETGID capability. This function is available only on POSIX platforms (i.e. not Windows, Android).
11	**initgroups(user, extra_group)** Reads /etc/group and initializes the group access list, using all the groups of which the user is a member. This is a privileged operation, which implies that you have to be at the root or have the CAP_SETGID capability. This function is available only 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 main.js with the following code:

```
// 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(process.memoryUsage());
```

Now run the main.js to see the result:

```
$ node main.js
```

Verify the Output while running your program on Linux machine.

```
Current directory: /web/com/1427106219_25089
Current version: v0.10.33
{ rss: 11505664, heapTotal: 4083456, heapUsed: 2157704 }
```

13. Utility Modules

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

S. No.	Module Name & Description
1	**OS Module** Provides basic operating-system related utility functions.
2	**Path Module** Provides utilities for handling and transforming file paths.
3	**Net Module** Provides both servers and clients as streams. Acts as a network wrapper.
4	**DNS Module** Provides functions to do actual DNS lookup as well as to use underlying operating system name resolution functionalities.
5	**Domain Module** Provides ways to handle multiple different I/O operations as a single group.

OS Module

Node.js **os** module provides a few basic operating-system related utility functions. This module can be imported using the following syntax.

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

Methods

S. 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

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

Example

The following example demonstrates a few OS methods. Create a js file named main.js with the following code.

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


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

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

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

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

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

Now run the main.js to see the result:

```
$ node main.js
```

Verify the Output.

```
endianness : LE
type : Linux
platform : linux
total memory : 25103400960 bytes.
free memory : 20676710400 bytes.
```

Path Module

Node.js **path** module is used for handling and transforming file paths. This module can be imported using the following syntax.

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

Methods

S.No.	Method & Description
1	**path.normalize(p)** Normalize a string path, taking care of '..' and '.' parts.
2	**path.join([path1][, path2][, ...])** Join all the arguments together and normalize the resulting path.

3	**path.resolve([from ...], to)** Resolves to an absolute path.
4	**path.isAbsolute(path)** Determines whether the 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.

Properties

S. 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.

Example

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

```
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('main.js'));


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

Now run the main.js to see the result:

```
$ node main.js
```

Verify the Output.

```
normalization : /test/test1/2slashes/1slash
joint path : /test/test1/2slashes/1slash
resolve : /web/com/1427176256_27423/main.js
ext name : .js
```

Net Module

Node.js **net** module is used to create both servers and clients. This module provides an asynchronous network wrapper and it can be imported using the following syntax.

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

Methods

S. 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 a listener for the 'connect' event. It 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 a listener for the 'connect' event. It 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 a listener for the 'connect' event. It is 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 a listener for the 'connect' event. It is a factory method which returns a new 'net.Socket'.
8	**net.isIP(input)** Tests if the input is an IP address. Returns 0 for invalid strings, 4 for IP version 4 addresses, and 6 for IP version 6 addresses.
9	**net.isIPv4(input)** Returns true if the input is a version 4 IP address, otherwise returns false.
10	**net.isIPv6(input)** Returns true if the 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

S.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. It 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, then 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, then calling the 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

S. No.	Events & 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 the 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.

S. No.	Events & 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 indicates if the socket was closed due to a transmission error.

Properties

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

S. 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

S. 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.
3	**socket.connect(path[, 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.
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. Necessary only 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 does 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 the 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, then 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, then calling ref again will have no effect.

Example

Create a js file named server.js with the following code:

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:

```
$ node server.js
```

Verify the Output.

```
server is listening
```

Create a js file named client.js with the following code:

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 from another terminal to see the result:

```
$ node client.js
```

Verify the Output.

```
connected to server!
Hello World!


disconnected from server
```

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

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

DNS Module

Node.js **dns** module is used to do actual DNS lookup as well as to use underlying operating system name resolution functionalities. This module provides an asynchronous network wrapper and can be imported using the following syntax.

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

Methods

S. 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, they 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 main.js with the following code:

```
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 main.js to see the result:

```
$ node main.js
```

Verify the Output.

```
address: 173.194.46.83
reverse for 173.194.46.83: ["ord08s11-in-f19.1e100.net"]
```

Domain Module

Node.js **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 emitter is executing its code within the run method of a domain.
- **External Binding** - Error emitter is added explicitly to a domain using its add method.

This module can be imported using the following syntax.

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

The 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 the following syntax:

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

Methods

S. No.	Method & Description

1	**domain.run(function)** Run the supplied function in the context of the domain, implicitly binding all event emitters, timers, and low-level 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. Whenever the execution switches 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.
8	**domain.dispose()** Once dispose has been called, the domain will no longer be used by callbacks bound into the domain via run, bind, or intercept, and a dispose event is emit

Properties

S. 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 main.js with the following code:

```
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 main.js to see the result:

```
$ node main.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\main.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
```

14. Web Module

What is a Web Server?

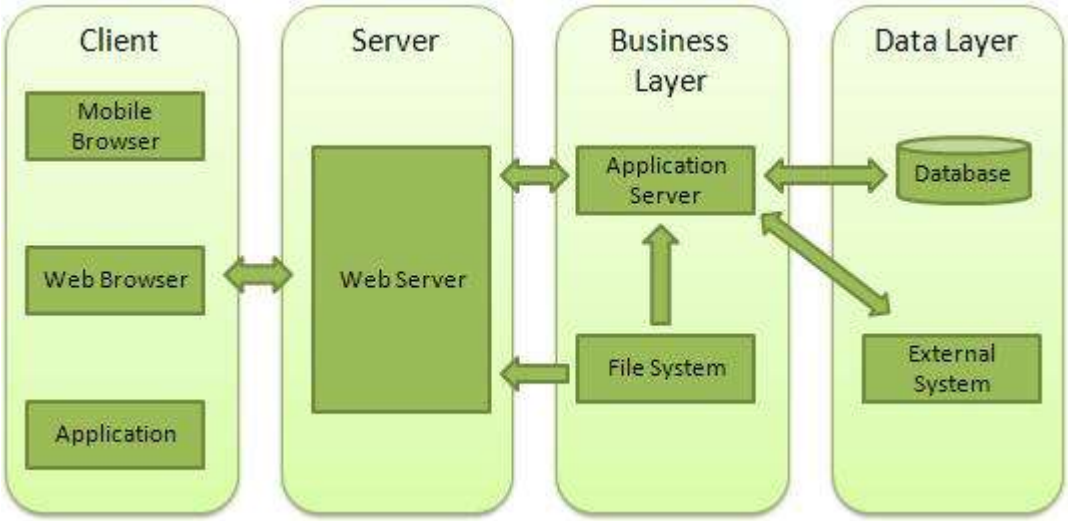
A Web Server is a software application which handles HTTP requests sent by the HTTP client, like web browsers, and returns web pages in response to the clients. Web servers usually deliver html documents along with images, style sheets, and scripts.

Most of the web servers support server-side scripts, using scripting languages or redirecting the task to an application server which retrieves data from a database and performs complex logic and then sends a result to the HTTP client through the Web server.

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

Web Application Architecture

A Web application is usually divided into four layers:



- **Client** - This layer consists of web browsers, mobile browsers or applications which can make HTTP requests to the web server.
- **Server** - This layer has the Web server which can intercept the requests made by the clients and pass them the response.
- **Business** - This layer contains the application server which is utilized by the web server to do the required processing. This layer interacts with the data layer via the database or some external programs.
- **Data** - This layer contains the databases or any other source of data.

Creating a Web Server using Node

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

Create a js file named server.js:

File: server.js

```
var http = require('http');
var fs = require('fs');
var url = require('url');



// Create a server
http.createServer( function (request, response) {
   // Parse the request containing file name
   var pathname = url.parse(request.url).pathname;

   // Print the name of the file for which request is made.
   console.log("Request for " + pathname + " received.");

   // Read the requested file content from file system
   fs.readFile(pathname.substr(1), function (err, data) {
      if (err) {
         console.log(err);
         // HTTP Status: 404 : NOT FOUND
         // Content Type: text/plain
         response.writeHead(404, {'Content-Type': 'text/html'});
      }else{
         //Page found
         // HTTP Status: 200 : OK
         // Content Type: text/plain
         response.writeHead(200, {'Content-Type': 'text/html'});

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

```
   });
}).listen(8081);

// Console will print the message
console.log('Server running at http://127.0.0.1:8081/');
```

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

File: index.htm

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

Now let us run the server.js to see the result:

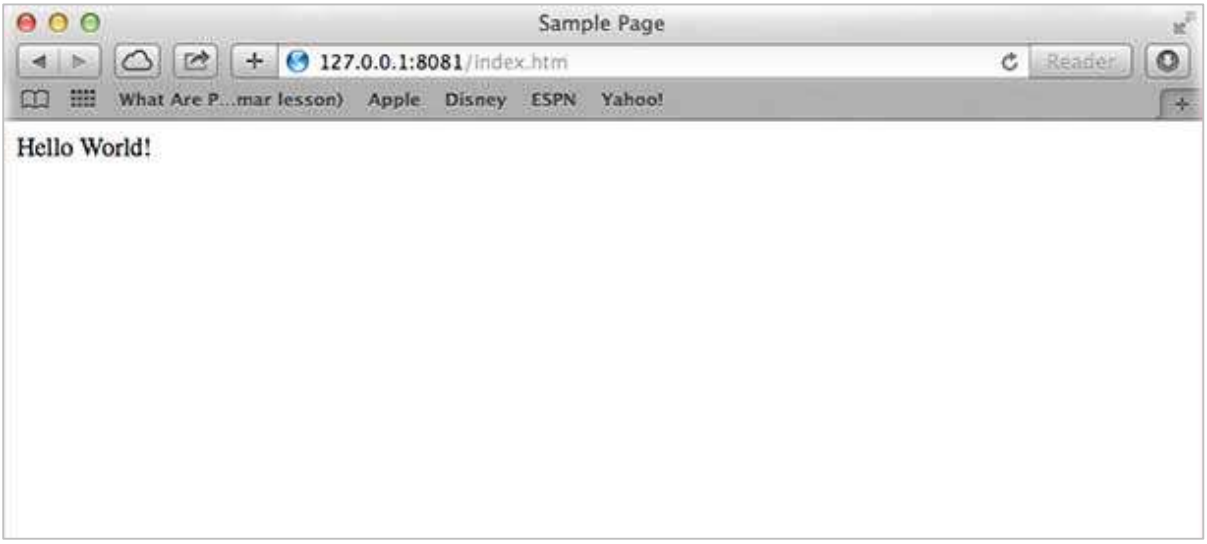
```
$ node server.js
```

Verify the Output.

```
Server running at http://127.0.0.1:8081/
```

Make a request to Node.js server

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



Verify the Output at the server end.

```
Server running at http://127.0.0.1:8081/
Request for /index.htm received.
```

Creating a Web client using Node

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

Create a js file named client.js:

File: client.js

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

// Options to be used by request
var options = {
   host: 'localhost',
   port: '8081',
   path: '/index.htm'
};

// Callback function is used to deal with response
var callback = function(response){
   // Continuously update stream with data
   var body = '';
```

```
   response.on('data', function(data) {
      body += data;
   });

   response.on('end', function() {
      // Data received completely.
      console.log(body);
   });
}
// Make a request to the server
var req = http.request(options, callback);
req.end();
```

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

```
$ node client.js
```

Verify the Output.

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

Verify the Output at the server end.

```
Server running at http://127.0.0.1:8081/
Request for /index.htm received.
```

15. Express Framework

Express Overview

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

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

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

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

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

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

Hello world Example

Following is a very basic Express app which starts a server and listens on port 3000 for connection. This app responds with **Hello World!** for requests to the homepage. For every other path, it will respond with a **404 Not Found.**

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


app.get('/', function (req, res) {
   res.send('Hello World');
})

var server = app.listen(8081, function () {

  var host = server.address().address
  var port = server.address().port

  console.log("Example app listening at http://%s:%s", host, port)

})
```

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

```
$ node server.js
```

You will see the following output:

```
Example app listening at http://0.0.0.0:8081
```

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

Request & Response

Express application uses a callback function whose parameters are **request** and **response** objects.

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

- **Request Object** - The request object represents the HTTP request and has properties for the request query string, parameters, body, HTTP headers, and so on.

- **Response Object** - The response object represents the HTTP response that an Express app sends when it gets an HTTP request.

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

Request Object

The **req** object represents the HTTP request and has properties for the request query string, parameters, body, HTTP headers, and so on.

Request Object Properties

Following is the list of few properties associated with request object.

S. No.	Properties & Description
1	**req.app** This property holds a reference to the instance of the express application that is using the middleware.
2	**req.baseUrl** The URL path on which a router instance was mounted.
3	**req.body** Contains key-value pairs of data submitted in the request body. By default, it is undefined, and is populated when you use body-parsing middleware such as **body-parser**
4	**req.cookies** When using cookie-parser middleware, this property is an object that contains cookies sent by the request.
5	**req.fresh** Indicates whether the request is "fresh." It is the opposite of req.stale.
6	**req.hostname** Contains the hostname from the "Host" HTTP header.

7	**req.ip** The remote IP address of the request.
8	**req.ips** When the trust proxy setting is true, this property contains an array of IP addresses specified in the "X-Forwarded-For" request header.
9	**req.originalUrl** This property is much like req.url; however, it retains the original request URL, allowing you to rewrite req.url freely for internal routing purposes.
10	**req.params** An object containing properties mapped to the named route "parameters". For example, if you have the route /user/:name, then the "name" property is available as req.params.name. This object defaults to {}.
11	**req.path** Contains the path part of the request URL.
12	**req.protocol** The request protocol string, "http" or "https" when requested with TLS.
13	**req.query** An object containing a property for each query string parameter in the route.
14	**req.route** The currently-matched route, a string.
15	**req.secure** A Boolean that is true if a TLS connection is established.
16	**req.signedCookies** When using cookie-parser middleware, this property contains signed cookies sent by the request, unsigned and ready for use.
17	**req.stale** Indicates whether the request is "stale," and is the opposite of req.fresh.
18	**req.subdomains** An array of subdomains in the domain name of the request.
19	**req.xhr** A Boolean value that is true if the request's "X-Requested-With" header field is "XMLHttpRequest", indicating that the request was issued by a client library such as jQuery

Request Object Methods

req.accepts(types)

```
req.accepts(types)
```

This method checks if the specified content types are acceptable, based on the request's Accept HTTP header field. Following are a few examples:

```
// Accept: text/html
req.accepts('html');
// => "html"

// Accept: text/*, application/json
req.accepts('html');
// => "html"
req.accepts('text/html');
// => "text/html"
```

req.get(field)

```
req.get(field)
```

This method returns the specified HTTP request header field. Following are a few examples:

```
req.get('Content-Type');
// => "text/plain"

req.get('content-type');
// => "text/plain"

req.get('Something');
// => undefined
```

req.is(type)

```
req.is(type)
```

This method returns true if the incoming request's "Content-Type" HTTP header field matches the MIME type specified by the type parameter. Following are a few examples:

```
// With Content-Type: text/html; charset=utf-8
req.is('html');
req.is('text/html');
req.is('text/*');
// => true
```

req.param(name [, defaultValue])

```
req.param(name [, defaultValue])
```

This method returns the value of param name when present. Following are few examples:

```
// ?name=tobi
req.param('name')
// => "tobi"

// POST name=tobi
req.param('name')
// => "tobi"

// /user/tobi for /user/:name
req.param('name')
// => "tobi"
```

Response Object

The **res** object represents the HTTP response that an Express app sends when it gets an HTTP request.

Response Object Properties

Following is the list of few properties associated with response object.

S. No.	Properties & Description
1	**res.app** This property holds a reference to the instance of the express application that is using the middleware.
2	**res.headersSent** Boolean property that indicates if the app sent HTTP headers for the response.
3	**res.locals** An object that contains response local variables scoped to the request.

Response Object Methods

res.append(field [, value])

```
res.append(field [, value])
```

This method appends the specified value to the HTTP response header field. Following are a few examples:

```
res.append('Link', ['<http://localhost/>', '<http://localhost:3000/>']);
res.append('Set-Cookie', 'foo=bar; Path=/; HttpOnly');
res.append('Warning', '199 Miscellaneous warning');
```

res.attachment([filename])

```
res.attachment([filename])
```

This method is used to send a file as an attachment in the HTTP response. Following are a few examples:

```
res.attachment('path/to/logo.png');
```

res.cookie(name, value [, options])

```
res.cookie(name, value [, options])
```

This method is used to set cookie name to value. The value parameter may be a string or object converted to JSON. Following are a few examples:

```
res.cookie('name', 'tobi', { domain: '.example.com', path: '/admin', secure:
true });

res.cookie('cart', { items: [1,2,3] });
res.cookie('cart', { items: [1,2,3] }, { maxAge: 900000 });
```

res.clearCookie(name [, options])

```
res.clearCookie(name [, options])
```

This method is used to clear the cookie specified by name. Following are a few examples:

```
res.cookie('name', 'tobi', { path: '/admin' });
res.clearCookie('name', { path: '/admin' });
```

res.download(path [, filename] [, fn])

```
res.download(path [, filename] [, fn])
```

This method is used to transfer the file at path as an "attachment". Typically, browsers will prompt the user for download. Following are a few examples:

```
res.download('/report-12345.pdf');

res.download('/report-12345.pdf', 'report.pdf');
```

```
res.download('/report-12345.pdf', 'report.pdf', function(err){

});
```

res.end([data] [, encoding])

```
res.end([data] [, encoding])
```

This method is used to end the response process. Following are a few examples:

```
res.end();

res.status(404).end();
```

res.format(object)

```
res.format(object)
```

This method is used to perform content-negotiation on the Accept HTTP header on the request object, when present. Following are a few examples:

```
res.format({
  'text/plain': function(){
    res.send('hey');
  },

  'text/html': function(){
    res.send('
hey
');
  },

  'application/json': function(){
    res.send({ message: 'hey' });
  },

  'default': function() {
    // log the request and respond with 406
    res.status(406).send('Not Acceptable');
  }
```

```
});
```

res.get(field)

```
res.get(field)
```

This method is used to return the HTTP response header specified by field. Here is an example:

```
res.get('Content-Type');
```

res.json([body])

```
res.json([body])
```

This method is used to send a JSON response. Following are a few examples:

```
res.json(null)
res.json({ user: 'tobi' })
res.status(500).json({ error: 'message' })
```

res.jsonp([body])

```
res.jsonp([body])
```

This method is used to send a JSON response with JSONP support. Following are a few examples:

```
res.jsonp(null)
res.jsonp({ user: 'tobi' })
res.status(500).jsonp({ error: 'message' })
```

res.links(links)

```
res.links(links)
```

This method is used to join the links provided as properties of the parameter to populate the response's Link HTTP header field. Following are a few examples:

```
res.links({
  next: 'http://api.example.com/users?page=2',
  last: 'http://api.example.com/users?page=5'
});
```

res.location(path)

```
res.location(path)
```

This method is used to set the response Location HTTP header field based on the specified path parameter. Following are a few examples:

```
res.location('/foo/bar');
res.location('foo/bar');
res.location('http://example.com');
```

res.redirect([status,] path)

```
res.redirect([status,] path)
```

This method is used to redirect to the URL dervied from the specified path, with specified HTTP status code status. Following are a few examples:

```
res.redirect('/foo/bar');
res.redirect('http://example.com');
res.redirect(301, 'http://example.com');
```

res.render(view [, locals] [, callback])

```
res.render(view [, locals] [, callback])
```

This method is used to render a view and sends the rendered HTML string to the client. Following are a few examples:

```
// send the rendered view to the client
res.render('index');

// pass a local variable to the view
res.render('user', { name: 'Tobi' }, function(err, html) {
  // ...
});
```

res.send([body])

```
res.send([body])
```

This method is used to send the HTTP response. Following are a few examples:

```
res.send(new Buffer('whoop'));
res.send({ some: 'json' });
res.send('
```

```
some html
');
```

res.sendFile(path [, options] [, fn])

```
res.sendFile(path [, options] [, fn])
```

This method is used to transfer the file at the given path. Sets the Content-Type response HTTP header field based on the filename's extension. Here is an example:

```
res.sendFile(fileName, options, function (err) {
  // ...
});
```

res.sendStatus(statusCode)

```
res.sendStatus(statusCode)
```

This method is used to set the response HTTP status code to statusCode and send its string representation as the response body. Following are a few examples:

```
res.sendStatus(200); // equivalent to res.status(200).send('OK')
res.sendStatus(403); // equivalent to res.status(403).send('Forbidden')
res.sendStatus(404); // equivalent to res.status(404).send('Not Found')
res.sendStatus(500); // equivalent to res.status(500).send('Internal Server
Error')
```

res.set(field [, value])

```
res.set(field [, value])
```

This method is used to set the response's HTTP header field to value. Following are a few examples:

```
res.set('Content-Type', 'text/plain');
res.set({
  'Content-Type': 'text/plain',
  'Content-Length': '123',
  'ETag': '12345'
})
```

res.status(code)

```
res.status(code)
```

This method is used to set the HTTP status for the response. Following are a few examples:

```
res.status(403).end();
res.status(400).send('Bad Request');
res.status(404).sendFile('/absolute/path/to/404.png');
```

res.type(type)

```
res.type(type)
```

This method is used to set the Content-Type HTTP header to the MIME type. Following are a few examples:

```
res.type('.html');              // => 'text/html'
res.type('html');               // => 'text/html'
res.type('json');               // => 'application/json'
res.type('application/json');   // => 'application/json'
res.type('png');                // => image/png:
```

Basic Routing

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

We will extend our Hello World program to handle more types of HTTP requests.

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

// This responds with "Hello World" on the homepage
app.get('/', function (req, res) {
   console.log("Got a GET request for the homepage");
   res.send('Hello GET');
})

// This responds a POST request for the homepage
app.post('/', function (req, res) {
   console.log("Got a POST request for the homepage");
```

```
   res.send('Hello POST');
})

// This responds a DELETE request for the /del_user page.
app.delete('/del_user', function (req, res) {
   console.log("Got a DELETE request for /del_user");
   res.send('Hello DELETE');
})

// This responds a GET request for the /list_user page.
app.get('/list_user', function (req, res) {
   console.log("Got a GET request for /list_user");
   res.send('Page Listing');
})

// This responds a GET request for abcd, abxcd, ab123cd, and so on
app.get('/ab*cd', function(req, res) {
   console.log("Got a GET request for /ab*cd");
   res.send('Page Pattern Match');
})


var server = app.listen(8081, function () {

  var host = server.address().address
  var port = server.address().port

  console.log("Example app listening at http://%s:%s", host, port)

})
```

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

```
$ node server.js
```

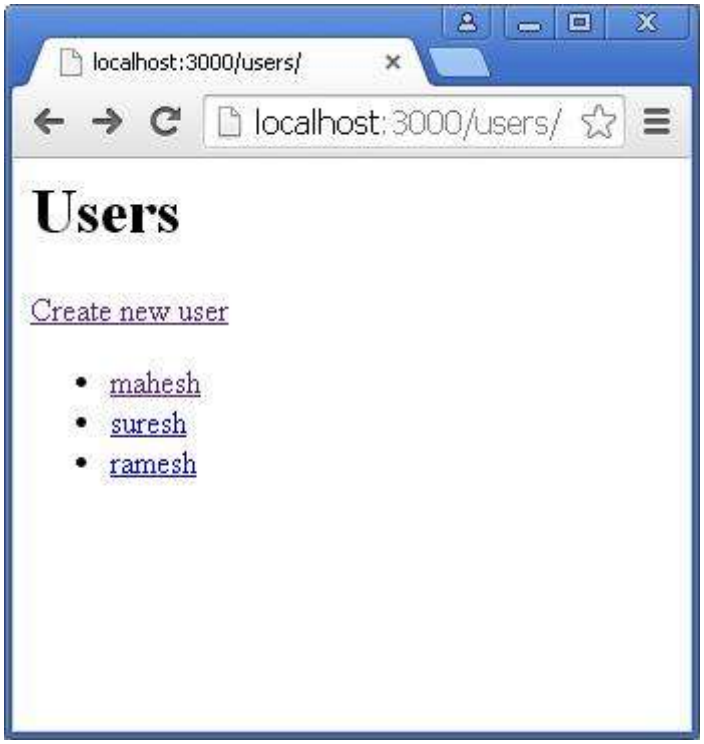
You will see the following output:

```
Example app listening at http://0.0.0.0:8081
```

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

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

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



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

Serving Static Files

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

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

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

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

```
node_modules
server.js
public/
public/images
public/images/logo.png
```

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

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

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

app.get('/', function (req, res) {
   res.send('Hello World');
})
```

```
var server = app.listen(8081, function () {

  var host = server.address().address
  var port = server.address().port

  console.log("Example app listening at http://%s:%s", host, port)

})
```

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

```
$ node server.js
```

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

GET Method

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

```
<html>
<body>
<form action="http://127.0.0.1:8081/process_get" method="GET">
First Name: <input type="text" name="first_name">  <br>

Last Name: <input type="text" name="last_name">
<input type="submit" value="Submit">
```

```
</form>
</body>
</html>
```

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

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

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

app.get('/index.htm', function (req, res) {
   res.sendFile( __dirname + "/" + "index.htm" );
})

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

   // Prepare output in JSON format
   response = {
       first_name:req.query.first_name,
       last_name:req.query.last_name
   };
   console.log(response);
   res.end(JSON.stringify(response));
})

var server = app.listen(8081, function () {

  var host = server.address().address
  var port = server.address().port

  console.log("Example app listening at http://%s:%s", host, port)

})
```

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

Now you can enter the First and Last Name and then click submit button to see the result and it should return the following result:

```
{"first_name":"John","last_name":"Paul"}
```

POST Method

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

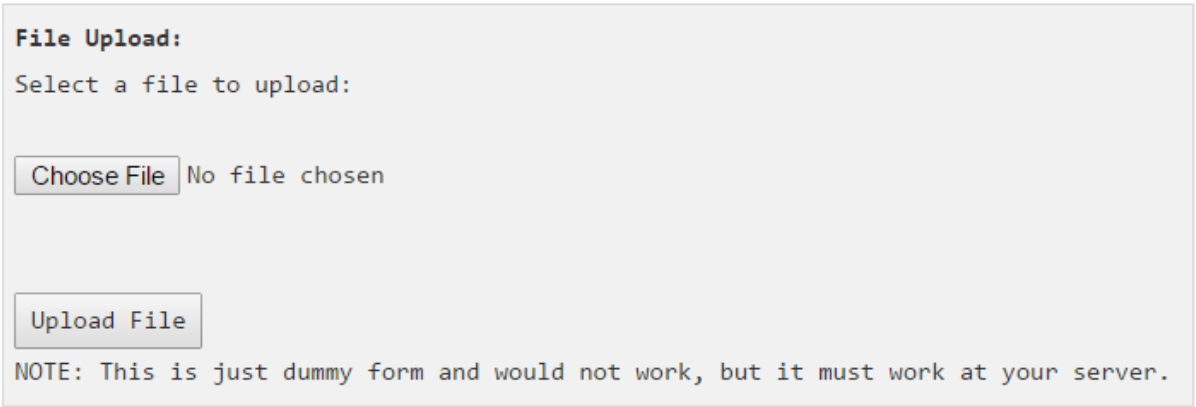
```
<html>
<body>
<form action="http://127.0.0.1:8081/process_post" method="POST">
First Name: <input type="text" name="first_name">  <br>

Last Name: <input type="text" name="last_name">
<input type="submit" value="Submit">
</form>
</body>
</html>
```

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

```
var express = require('express');
var app = express();
var bodyParser = require('body-parser');

// Create application/x-www-form-urlencoded parser
var urlencodedParser = bodyParser.urlencoded({ extended: false })

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

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

```
   res.sendFile( __dirname + "/" + "index.htm" );
})

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

   // Prepare output in JSON format
   response = {
       first_name:req.body.first_name,
       last_name:req.body.last_name
   };
   console.log(response);
   res.end(JSON.stringify(response));
})

var server = app.listen(8081, function () {

  var host = server.address().address
  var port = server.address().port

  console.log("Example app listening at http://%s:%s", host, port)

})
```

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



Now you can enter the First and Last Name and then click the submit button to see the following result:

```
{"first_name":"John","last_name":"Paul"}
```

File Upload

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

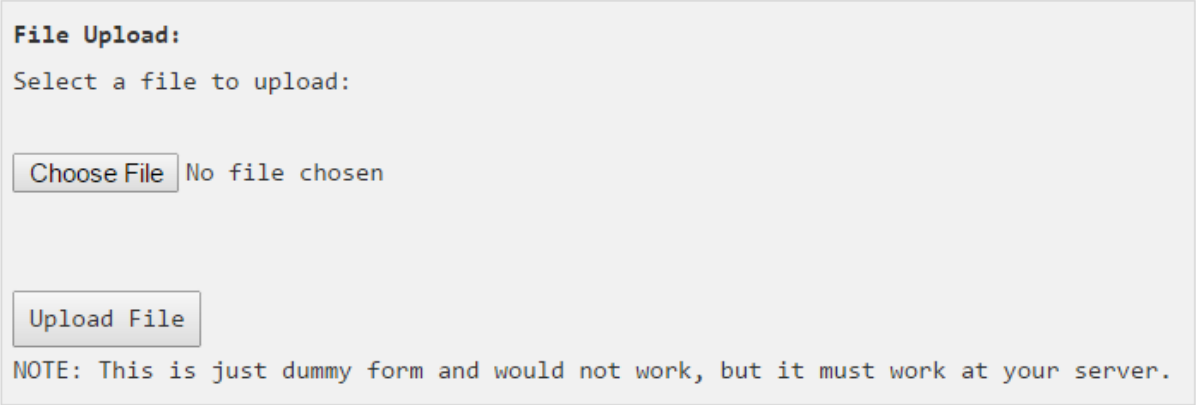
```
<html>
<head>
<title>File Uploading Form</title>
</head>
<body>
<h3>File Upload:</h3>
Select a file to upload: <br />
<form action="http://127.0.0.1:8081/file_upload" method="POST"
      enctype="multipart/form-data">
<input type="file" name="file" size="50" />
<br />
<input type="submit" value="Upload File" />
</form>
</body>
</html>
```

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

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

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

app.use(express.static('public'));
app.use(bodyParser.urlencoded({ extended: false }));
app.use(multer({ dest: '/tmp/'}));

app.get('/index.htm', function (req, res) {
   res.sendFile( __dirname + "/" + "index.htm" );
})

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

```
   console.log(req.files.file.name);
   console.log(req.files.file.path);
   console.log(req.files.file.type);

   var file = __dirname + "/" + req.files.file.name;
   fs.readFile( req.files.file.path, function (err, data) {
        fs.writeFile(file, data, function (err) {
         if( err ){
              console.log( err );
         }else{
               response = {
                   message:'File uploaded successfully',
                   filename:req.files.file.name
              };
          }
          console.log( response );
          res.end( JSON.stringify( response ) );
       });
   });
})

var server = app.listen(8081, function () {

  var host = server.address().address
  var port = server.address().port

  console.log("Example app listening at http://%s:%s", host, port)

})
```

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



Cookies Management

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

```
var express      = require('express')
var cookieParser = require('cookie-parser')

var app = express()
app.use(cookieParser())

app.get('/', function(req, res) {
  console.log("Cookies: ", req.cookies)
})

app.listen(8081)
```

16. RESTful API

What is REST Architecture?

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

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

HTTP methods

The following four HTTP methods are commonly used in REST based architecture.

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

RESTful Web Services

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

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

Creating RESTful for a Library

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

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

```
        "password" : "password1",
        "profession" : "teacher",
        "id": 1
    },
    "user2" : {
       "name" : "suresh",
        "password" : "password2",
        "profession" : "librarian",
        "id": 2
    },
    "user3" : {
       "name" : "ramesh",
        "password" : "password3",
        "profession" : "clerk",
        "id": 3
    }
}
```

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

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

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

List Users

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

server.js

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

app.get('/listUsers', function (req, res) {
   fs.readFile( __dirname + "/" + "users.json", 'utf8', function (err, data) {
       console.log( data );
       res.end( data );
   });
})

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

})
```

Now try to access the defined API using *http://127.0.0.1:8081/listUsers* on your local machine. It should produce following result:

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

```
{
   "user1" : {
      "name" : "mahesh",
      "password" : "password1",
      "profession" : "teacher",
      "id": 1
   },
   "user2" : {
      "name" : "suresh",
      "password" : "password2",
      "profession" : "librarian",
```

```
      "id": 2
   },
   "user3" : {
      "name" : "ramesh",
      "password" : "password3",
      "profession" : "clerk",
      "id": 3
   }
}
```

Add Users

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

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

You can accept the same input in the form of JSON using Ajax call but for demonstration purpose, we are hard-coding it here. Following is the **addUser** API to a new user in the database:

server.js

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

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

```
   }
}

app.post('/addUser', function (req, res) {
   // First read existing users.
   fs.readFile( __dirname + "/" + "users.json", 'utf8', function (err, data) {
       data = JSON.parse( data );
       data["user4"] = user["user4"];
       console.log( data );
       res.end( JSON.stringify(data));
   });
})

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

Now try to access defined API using **URL: http://127.0.0.1:8081/addUser** and **HTTP Method : POST** on local machine using any REST client. This should produce following result:

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

Show Detail

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

server.js

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

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

var server = app.listen(8081, function () {

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

})
```

Now try to access defined API using **URL: http://127.0.0.1:8081/2** and **HTTP Method : GET** on local machine using any REST client. This should produce following result:

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

Delete a User

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

server.js

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

var id = 2;

app.delete('/deleteUser', function (req, res) {
```

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

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

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

Now try to access defined API using **URL: http://127.0.0.1:8081/deleteUser** and **HTTP Method : DELETE** on local machine using any REST client. This should produce the following result:

```
{"user1":{"name":"mahesh","password":"password1","profession":"teacher","id":1},
"user3":{"name":"ramesh","password":"password3","profession":"clerk","id":3}}
```

17. Scaling an Application

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

Child processes always have three streams **child.stdin**, **child.stdout**, and **child.stderr** which may be shared with the stdio streams of the parent process.

Node provides **child_process** module which has the following three major ways to create a child process.

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

The exec() method

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

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

Parameters

Here is the description of the parameters used:

- **command** (String) The command to run, with space-separated arguments
- **options** (Object) may comprise one or more of the following options:
    - **cwd** (String) Current working directory of the child process
    - **env** (Object) Environment key-value pairs
    - **encoding** (String) Default: 'utf8'
    - **shell** (String) Shell to execute the command with. Default: '/bin/sh' on UNIX, 'cmd.exe' on Windows. The shell should understand the -c switch on UNIX or /s /c on Windows. On Windows, command line parsing should be compatible with cmd.exe.
    - **timeout** (Number) Default: 0
    - **maxBuffer** (Number) Default: 200*1024

- **killSignal** (String) Default: 'SIGTERM'
- **uid** (Number) Sets the user identity of the process.
- **gid** (Number) Sets the group identity of the process.

- **callback** The function gets three arguments **error**, **stdout**, and **stderr** which are called with the output when the process terminates.

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

Example

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

File: support.js

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

File: master.js

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

for(var i=0; i<3; i++) {
   var workerProcess = child_process.exec('node support.js '+i,
      function (error, stdout, stderr) {
         if (error) {
            console.log(error.stack);
            console.log('Error code: '+error.code);
            console.log('Signal received: '+error.signal);
         }
         console.log('stdout: ' + stdout);
         console.log('stderr: ' + stderr);
      });

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

Now run the master.js to see the result:

```
$ node master.js
```

Verify the Output. Server has started.

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

stderr:
Child process exited with exit code 0
stdout: Child Process 0 executed.

stderr:
Child process exited with exit code 0
stdout: Child Process 2 executed.
```

The spawn() Method

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

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

Parameters

Here is the description of the parameters used:

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

- **gid** (Number) Sets the group identity of the process.

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

Example

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

File: support.js

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

File: master.js

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

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

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

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

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

Now run the master.js to see the result:

```
$ node master.js
```

Verify the Output. Server has started

```
stdout: Child Process 0 executed.


child process exited with code 0
```

```
stdout: Child Process 1 executed.


stdout: Child Process 2 executed.


child process exited with code 0
child process exited with code 0
```

The fork() Method

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

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

Parameters

Here is the description of the parameters used:

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

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

Example

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

File: support.js

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

File: master.js

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

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

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

Now run the master.js to see the result:

```
$ node master.js
```

Verify the Output. Server has started.

```
Child Process 0 executed.
Child Process 1 executed.
Child Process 2 executed.
child process exited with code 0
child process exited with code 0
child process exited with code 0
```

18. Packaging

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

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

JXcore Installation

Installing JXcore is quite simple. Here we have provided step-by-step instructions on how to install JXcore on your system. Follow the steps given below:

Step 1

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

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

Step 2

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

```
$ unzip jx_rh64.zip
$ cp jx_rh64/jx /usr/bin
```

Step 3

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

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

Step 4

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

```
$ jx --version
v0.10.32
```

Packaging the Code

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

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

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

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

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

- **index.jxp** This is an intermediate file which contains the complete project detail needed to compile the project.

- **index.jx** This is the binary file having the complete package that is ready to be shipped to your client or to your production environment.

Launching JX File

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

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

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

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

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