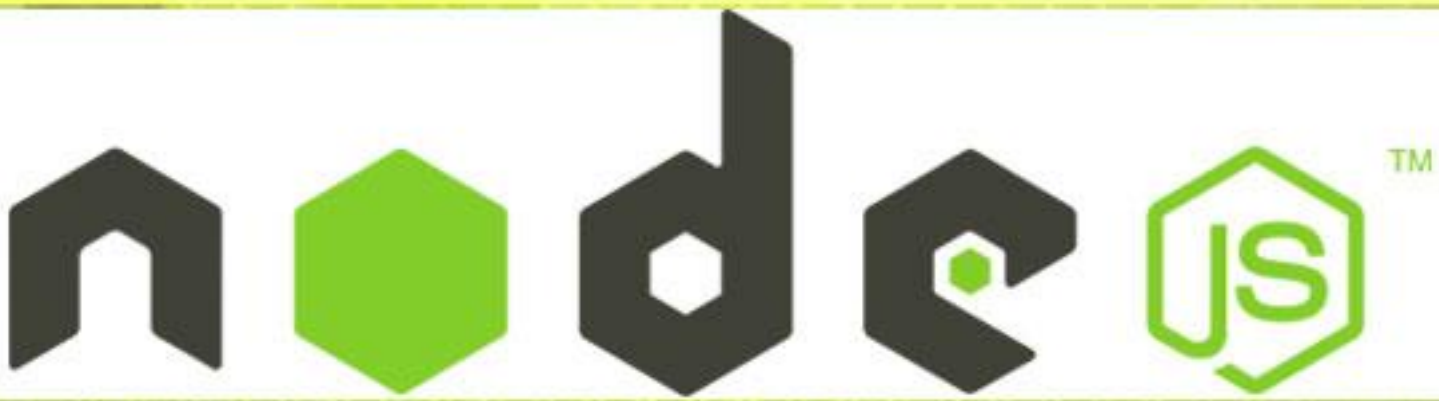


tutorialspoint

SIMPLYEASYLEARNING

www.tutorialspoint.com

https://www.facebook.com/tutorialspointindia

https://twitter.com/tutorialspoint

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**.

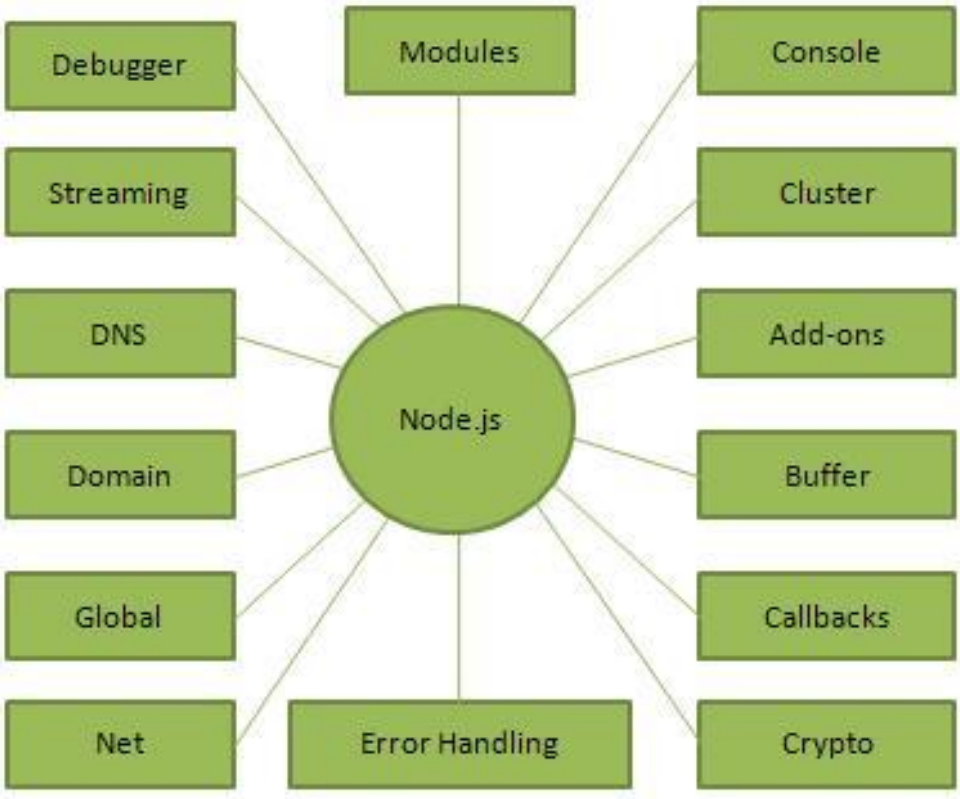
Who Uses Node.js?

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.

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
```

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"
```


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.

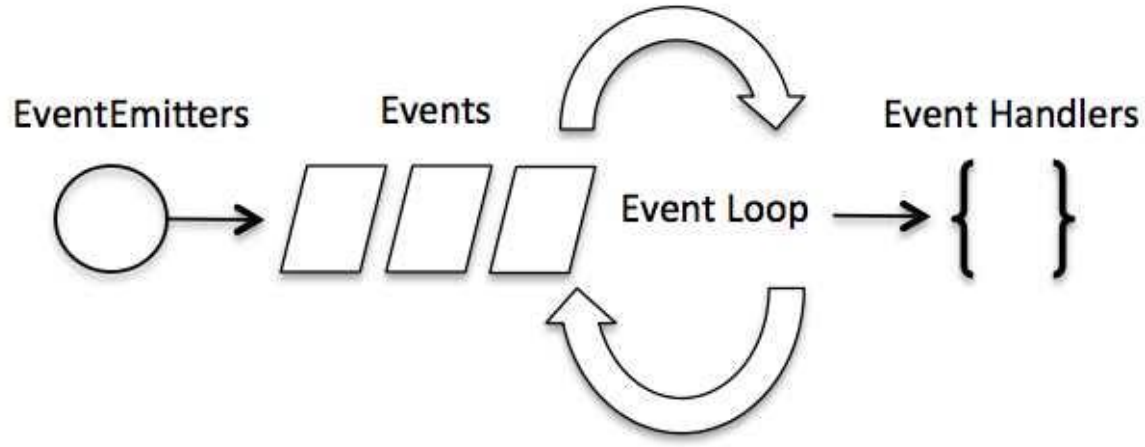
7. EVENT LOOP

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
```

```
   }
   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)**