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.

Node.js is a platform built on [Chrome's JavaScript runtime](https://code.google.com/p/v8/) for easily building fast and scalable network applications. Node.js uses an event-driven, non-blocking I/O model that makes it lightweight and efficient, perfect for data-intensive real-time applications that run across distributed devices.

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

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

Node.js = Runtime Environment + JavaScript Library

Features of Node.js

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

* **Asynchronous and Event Driven** − All APIs of Node.js library are asynchronous, that is, non-blocking. It essentially means a Node.js based server never waits for an API to return data. The server moves to the next API after calling it and a notification mechanism of Events of Node.js helps the server to get a response from the previous API call.
* **Very Fast** − Being built on Google Chrome's V8 JavaScript Engine, Node.js library is very fast in code execution.
* **Single Threaded but Highly Scalable** − Node.js uses a single threaded model with event looping. Event mechanism helps the server to respond in a non-blocking way and makes the server highly scalable as opposed to traditional servers which create limited threads to handle requests. Node.js uses a single threaded program and the same program can provide service to a much larger number of requests than traditional servers like Apache HTTP Server.
* **No Buffering** − Node.js applications never buffer any data. These applications simply output the data in chunks.

Who Uses Node.js?

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

* [Projects, Applications, and Companies Using Node](https://github.com/joyent/node/wiki/projects,-applications,-and-companies-using-node)

Concepts

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



Where to Use Node.js?

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

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

 is not in the callback function. All the applications code that is inside callback functions will run in the event loop. EventLoop is the heart of node.js. When we start our node application the event loop starts running right away. Most of the work is done in the event loop.

Nodejs use event-driven-architecture.

* Events are emitted.
* Event loop picks them up.
* Callbacks are called.

# Explain the working of Node.js

Node.js is an open-source backend javascript runtime environment. It is a used as backend service where javascript works on the server-side of the application. This way javascript is used on both frontend and backend. Node.js runs on chrome v8 engine which converts javascript code into machine code, it is highly scalable, lightweight, fast, and data-intensive.

**Working of Node.js:**Node.js accepts the request from the clients and sends the response, while working with the request node.js handles them with a single thread. To operate I/O operations or requests node.js use the concept of threads. Thread is a sequence of instructions that the server needs to perform. It runs parallel on the server to provide the information to multiple clients. Node.js is an event loop single-threaded language. It can handle concurrent requests with a single thread without blocking it for one request.

Node.js basically works on two concept

* Asynchronous
* Non-blocking I/O

**Non-blocking I/o:**Non-blocking i/o  means working with multiple requests without blocking the thread for a single request. I/O basically interacts with external systems such as files, databases. Node.js is not used for CPU-intensive work means for calculations, video processing because a single thread cannot handle the CPU works.

**Asynchronous:**Asynchronous is executing a callback function. The moment we get the response from the other server or database it will execute a callback function. Callback functions are called as soon as some work is finished and this is because the node.js uses an event-driven architecture. The single thread doesn’t work with the request instead it sends the request to another system which resolves the request and it is accessible for another request.

To implement the concept of the system to handle the request  node.js uses the concept of Libuv.

Libuv is an open-source library built-in c++. It has a strong focus on asynchronous and  I/O, this gives node access to the underlying computer operating system, file system, and networking.

Libuv implements two extremely important features of node.js

* Event loop
* Thread pool

**Event loop:**The event loop contains a single thread and is responsible for handling easy tasks like executing callbacks and network I/O. When the program is to initialize all the top-level code is executed, the code is not in the callback function. All the applications code that is inside callback functions will run in the event loop. EventLoop is the heart of node.js. When we start our node application the event loop starts running right away. Most of the work is done in the event loop.

Nodejs use event-driven-architecture.

* Events are emitted.
* Event loop picks them up.
* Callbacks are called.

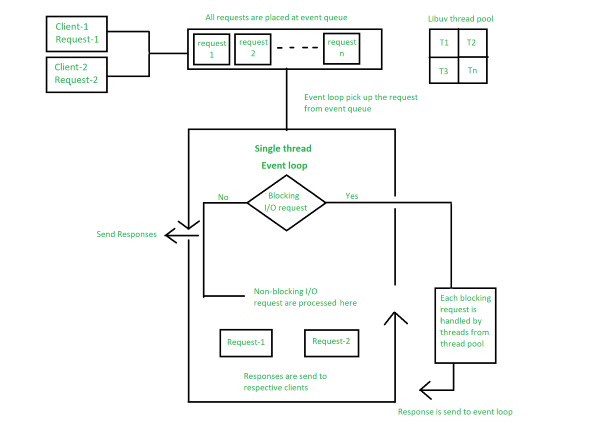
**Event queue:**As soon as the request is sent the thread places the request into a queue. It is known as an event queue. The process like app receiving HTTP request or server or a timer will emit event as soon as they are done with the work and event loop will pick up these events and call the callback functions that are associated with each event and response is sent to the client.

The event loop is an indefinite loop that continuously receives the request and processes them. It checks the queue and waits for the incoming request indefinitely.

**Thread pool:**Though node.js is single-threaded it internally maintains a thread pool. When non-blocking requests are accepted there are processed in an event loop, but while accepting blocking requests it checks for available threads in a thread pool, assigns a thread to the client’s request which is then processed and send back to the event loop, and response is sent to the respective client.

The thread pool size can be change:

process.env.UV\_THREADPOOL\_SIZE = 1;



## 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 **Live Demo** option available at the top right corner of the below sample code box (on our website) −

[Live Demo](http://tpcg.io/sTJjjt)

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

## Text Editor

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

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

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

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

## The Node.js Runtime

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

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

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

## Download Node.js archive

Download latest version of Node.js installable archive file from [Node.js Downloads](http://nodejs.org/download/). 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 Windows

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

## Verify installation: Executing a File

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

[Live Demo](http://tpcg.io/sTJjjt)

/\* Hello, World! program in node.js \*/

console.log("Hello, World!")

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

$ node main.js

If everything is fine with your installation, this should produce the following result −

Hello, World!

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 −

* **Import required modules** − We use the **require** directive to load Node.js modules.
* **Create server** − A server which will listen to client's requests similar to Apache HTTP Server.
* **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.

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](https://www.tutorialspoint.com/nodejs_terminal_online.php)

### **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 World

undefined

### **Multiline Expression**

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

$ node

> var x = 0

undefined

> do {

... x++;

... console.log("x: " + x);

... }

while ( x < 5 );

x: 1

x: 2

x: 3

x: 4

x: 5

undefined

>

**...** comes automatically when you press 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)

>