Node JS

Node.js is a server side platform built on google crome’s javascript engine. Node.js is build for easliy building fast, scalable network applications.

NodeJS:

* It’s just javascript
* Server side JS on googles VB
* Write scalable web apps
* Single threaded
* Everything is asynchronous
* All I/O is non-blocking

What we can do with NodeJS:

* JSONApi
* Single Page Apps
* Scalable web apps
* Interact with filesystems
* Datastreaming
* Heavy computations
* General scripting

What Node ISN’T

* Fullstack web framework
* MVC framework
* Pre-bundled with Database

Node.js is an open source, cross-platform Runtime Environment for 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.

**Features of Node.js:**

* **Aynchronous and Event Driven :** All APIs of Node.js library are aynchronous that is non-blocking. It essentially means a Node.js based server never waits for a API to return data. Server moves to next API after calling it and a notification mechanism of Events of Node.js helps server to get response from the previous API call.
* **Very Fast :** Node.js library is very fast in code execution.
* **Single Threaded but highly Scalable :** Node.js uses a single threaded program and same program can services much larger number of requests than traditional server like Apache HTTP Server.
* **No Buffering** - Node.js applications never buffer any data. These applications simply output the data in chunks.
* **License** - Node.js is released under the MIT license.

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

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

Environment Setup:

**Text Editor:** This will be used to type your program. Examples of few editors include Windows Notepad.

**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.

**Download Node.js archive:**

Download latest version of Node.js installable archive file from [www.nodejs.org](http://www.nodejs.org/) and install. Below are the versions available on different OS.

|  |  |
| --- | --- |
| OS | Version Name |
| Windows | node-v0.12.0-x64.msi |
| Linux | node-v0.12.0-linux-x86.tar.gz |
| Max | node-v0.12.0-darwin-x86.ta.gz |
| SunOS | node-v0.12.0-sunos-x86.tar.gz |

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.

We need to download & install git from [www.git-scm.com](http://www.git-scm.com/)

Simple example in node.js : create a js file names main.js having the following code.

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

console.log("Hello, World!")

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

$ node main.js

this should produce the following result:

Hello, World!

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

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

**Creating Node.js Application**

Step 1 - Import required module

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

var http = require("http");

Step 2 - Create Server

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

http.createServer(function (request, response) {

// Send the HTTP header

// HTTP Status: 200 : OK

// Content Type: text/plain

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

// Send the response body as "Hello World"

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

}).listen(8081);

// Console will print the message

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

Step 3: Testing Request & Response

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

var http = require("http");

http.createServer(function (request, response) {

// Send the HTTP header

// HTTP Status: 200 : OK

// Content Type: text/plain

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

// Send the response body as "Hello World"

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

}).listen(8081);

// Console will print the message

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

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

$ node main.js

Verify the Output. Server has started

Server running at <http://127.0.0.1:8081/>

Make a request to Node.js server

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

**REPL Terminal**

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

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

**Starting REPL :**

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

$ node

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

$ node

>

**Simple Example:**

$ node

> 1 + 3

4

> 1 + ( 2 \* 3 ) - 4

3

>

**Use Variables**

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

**Example:**

$ node

> x = 10

10

> var y = 10

undefined

> x + y

20

> console.log("Hello NodeJs 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 enters after opening bracket. Node automatically checks the continuity of expressions.

**Underscore Variable**

You can use undercore **\_** to get the last result:

$ node

> var x = 10

undefined

> var y = 20

undefined

> x + y

30

> var sum = \_

undefined

> console.log(sum)

30

undefined

>

**REPL Commands**

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

**Stopping REPL**

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

$ node

>

(^C again to quit)

>

**Node.js –(Node Package Manager) npm**

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

* Online repositories for node.js packages/modules which are searchable on [search. nodejs.org](http://search.nodejs.org/)
* Command line utility to install Node.js packages, do version management and dependency management of Node.js packages.

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

$ npm --version

2.7.1

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

$ sudo npm install npm -g

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

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

Installing Modules using npm: Here there is a simple syntax to install npm modules :

Syntax: $ 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');

Use following command To uninstall express.

$ npm uninstall express

Updating a module

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

$ npm update express

## Create a module

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

$ npm init

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

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

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

and exactly what they do.

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

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

Press ^C at any time to quit.

name: (webmaster)

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

$ npm adduser

Username: mcmohd

Password:

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

Now its time to publish your module:

$ npm publish

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

**What is Callback?**

A callback function is called at the completion of a given task. Callback is an asynchronous equivalent for a function.

For example, a function to read a file may start reading file and return the control to execution environment immidiately so that next instruction can be executed. Once file I/O is complete, it will call the callback function while passing the callback function, the content of the file as parameter.

Blocking Code Example:

Create one examplefile.txt file and put below content.

Callback is an asynchronous equivalent for a function. A callback function is called at the completion of a given task.

Create a js file named main.js which has the following code:

var fs = require("fs");

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

console.log(data.toString());

console.log("Ended");

Now run the main.js to see the result:

$ node main.js

Non-Blocking Code Example:

Please update below code in main.js file

var fs = require("fs");

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

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

console.log(data.toString());

});

console.log("Ended");

**Event Looping:**

Node js is a single threaded application but it support concurrency via concept of event and callbacks. As every API of Node js are asynchronous and being a single thread, it uses **async** function calls to maintain the concurrency.

**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.

**Passing arguments and this to listeners**

The eventEmitter.emit() method allows an arbitrary set of arguments to be passed to the listener functions. It is important to keep in mind that when an ordinary listener function is called by the EventEmitter, the standard this keyword is intentionally set to reference the EventEmitter to which the listener is attached.

Check event-loop-example.js for event looping

**How Node JS Application Works:** Any async function accepts a callback as a last parameter and the callback function accepts error as a first parameter.

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("Ended");

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

**Event Emitter**

All objects that emit events are instances of the EventEmitter class. These objects expose an eventEmitter.on() function that allows one or more Functions to be attached to named events emitted by the object.

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

EventEmitter class lies in **events** module.

Syntax:

// Import events module

var events = require('events');

// Create an eventEmitter object

var eventEmitter = new events.EventEmitter();

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

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

Methods:

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

**Class Methods:**

1. **listenerCount(emitter, event):** Return the number of listeners for a given event.

**Events:**

1. **newListener**:
2. **event**: - String The event name
3. **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.

1. **removeListener**:
2. **event**: - String The event name
3. **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:**

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.");

run the program by using below command:

$ node main.js

You will get output like below:

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.

**Buffers:** Buffer class is a global class and can be accessed in application without importing buffer module.

**Creating Buffers**: Node Buffer can be constructed in a variety of ways.

**Method1:** Syntax to create an uninitiated buffer of 10 octets:

var buf= new Buffer (10);

**Method2:** Create buffer from a given array.

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