**Table to Contents:**

**What s Node.js**

**Features of Node.js**

**Where to use Node.js?**

**Where to not use Node.js?**

## Creating Node.js Application

# Node.js - REPL Terminal

**Node.js expression**

**Package install locally**

**Package update and uninstall**

# Node.js - Callbacks Concept

**What is Node.js?**

Node.js is a server-side platform built on Google Chrome's JavaScript Engine (V8 Engine).

## 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.

## 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.

## Installation on Windows

## Verify installation: Executing a File

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

/\* create file with name node.js \*/

console.log("Hello, World - Tech!")

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 - Tech!

# 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 −

* **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.

# Node.js - 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 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)

>

## 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

## 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: test

Password:

Email: (this IS public) test@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.

# Node.js - Callbacks Concept

## What is Callback?

Callback is an asynchronous equivalent for a function. A callback function is called at the completion of a given task. Node makes heavy use of callbacks. All 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 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.

## Blocking Code Example

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

Test 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.

test 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.

Tech 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

Tech 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.