**Node Js**

Overview: JavaScript’s rising popularity has brought with it a lot of changes, and the face of web development today is dramatically different. The things that we can do on the web nowadays with JavaScript running on the server, as well as in the browser, were hard to imagine just several years ago, or were encapsulated within sandboxed environments like Flash or Java Applets.

Beyond that, it’s worth noting that Ryan Dahl, the creator of Node.js, was aiming to create real-time websites with push capability, “inspired by applications like Gmail”. In Node.js, he gave developers a tool for working in the non-blocking, event-driven I/O paradigm. After over 20 years of stateless-web based on the stateless request-response paradigm, we finally have web applications with real-time, two-way connections

**Module 1:- Introduction to Node JS**

Introduction to Node JS In this module, you learn what are Node JS and advantages of Node JS and How Node JS Works and the difference between the traditional web server and what are the limitations of the traditional web server modal.

**Introduction**

The modern web application has really come a long way over the years with the introduction of many popular frameworks such as bootstrap, Angular JS, etc. All of these frameworks are based on the popular[JavaScript](https://www.guru99.com/interactive-javascript-tutorials.html)framework.

But when it came to developing server-based applications, there was a kind of void, and this is where Node.js came into the picture.

Node.js is also based on the JavaScript framework, but it is used for developing server-based applications. While going through the entire tutorial, we will look into Node.js in detail and how we can use it to develop server-based applications.

**What is Node JS?**

Node.js is an open source server environment

Node.js is free

Node.js runs on various platforms (Windows, Linux, Unix, Mac OS X, etc.)

Node.js uses JavaScript on the server

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 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.

**Advantages of Node JS**

**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.

**Using Node.js to execute scripts**

**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

**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
* Node.js Process Model

**Where Not to Use Node.js?**

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

**Execution of node script.**

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

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

console.log("Hello, World!")

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

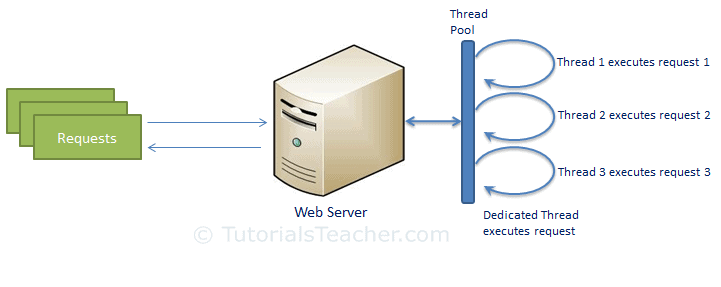
$ node main.js

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

**Node.js Process Model**

Traditional Web Server Model

In the traditional web server model, each request is handled by a dedicated thread from the thread pool. If no thread is available in the thread pool at any point of time then the request waits till the next available thread. Dedicated thread executes a particular request and does not return to thread pool until it completes the execution and returns a response.

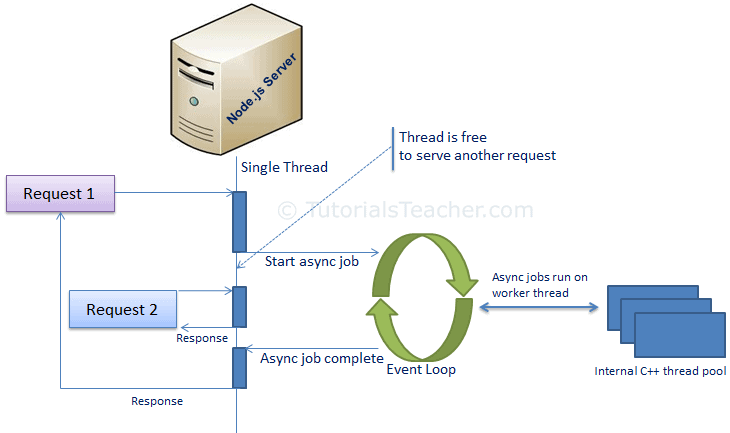


## Node.js Process Model

Node.js processes user requests differently when compared to a traditional web server model. Node.js runs in a single process and the application code runs in a single thread and thereby needs less resources than other platforms. All the user requests to your web application will be handled by a single thread and all the I/O work or long running job is performed asynchronously for a particular request. So, this single thread doesn't have to wait for the request to complete and is free to handle the next request. When asynchronous I/O work completes then it processes the request further and sends the response.

An event loop is constantly watching for the events to be raised for an asynchronous job and executing callback function when the job completes. Internally, Node.js uses [libev](http://software.schmorp.de/pkg/libev.html) for the event loop which in turn uses internal C++ thread pool to provide asynchronous I/O.

The following figure illustrates asynchronous web server model using Node.js.



Node.js process model increases the performance and scalability with a few caveats. Node.js is not fit for an application which performs CPU-intensive operations like image processing or other heavy computation work because it takes time to process a request and thereby blocks the single thread.

**Module 2:- Setup Dev Environment**

**Install Node.js**

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 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.

**Working in REPL**

The term REPL stands for Read Eval Print and Loop. It specifies a computer environment

Like a window console or a Unix/Linux shell where you can enter the commands and the

System responds with an output in an interactive mode.

**REPL Environment**

The Node.js or node come bundled with REPL environment.

Each part of the REPL environment has a specific work.

**Read:** It reads user's input; parse the input into JavaScript data-structure and stores in memory.

**Eval:** It takes and evaluates the data structure.

**Print:** It prints the result.

**Loop:** It loops the above command until user press ctrl-c twice.

**How to start REPL**

You can start REPL by simply running "node" on the command prompt.

You can execute various mathematical operations on REPL Node.js command prompt:

**Node.js Simple expressions**

After starting REPL node command prompt put any mathematical expression:

Example: >10+20-5

**Using variable**

Variables are used to store values and print later. If you don't use var keyword 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 using console.log ().

**Node.js Multiline expressions**

Node REPL supports multiline expressions like JavaScript. See the following do-while loop example:

var x = 0

undefined

> do {

... x++;

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

... } while ( x < 10 );

**Node.js REPL Commands**

Commands Description

ctrl + c It is used to terminate the current command.

ctrl + c twice It terminates the node repl.

ctrl + d It terminates the node repl.

up/down keys It is used to see command history and modify previous commands.

tab keys It specifies the list of current command.

.help It specifies the list of all commands.

.break It is used to exit from multi-line expressions.

.clear It is used to exit from multi-line expressions.

.save filename It saves current node repl session to a file.

.load filename It is used to load file content in current node repl session.

**Node.js Exit REPL**

Use ctrl + c command twice to come out of Node.js REPL.

**Node JS Console**

# Console Logging

If you start the Node.js binary without any arguments, you will see the REPL command prompt, the **>** character and from this prompt (interactive shell ) you can execute raw JavaScript code you wish. Those who are familiar with browser-side development have probably used console.log for writing information to the console and debugging purposes.

Similar to the Node.js, there is a built-in console object with several methods which works for printing to stdout and stderr.

List of Methods

* **console.log([data], [...])**
* **console.info([data], [...])**
* **console.error([data], [...])**
* **console.warn([data], [...])**
* **console.dir(obj)**
* **console.time(label)**
* **console.timeEnd(label)**
* **console.trace(label)**
* **console.assert(expression, [message])**

**console.log([data], [...])**

The console.log() method is used to print to stdout with newline. Like printf() the function can take multiple arguments.

**console.info([data], [...])**

The method is same as console.log.

**console.error([data], [...])**

The console.error() method works the same as console.log, except that the output is sent to stderr instead of stdout. As stderr is always written to synchronously, therefore in node.js any use of console.error, or other functions that write to stderr, will block your process until the output has all been written. The method is useful for error messages, but excessive use could slow down your process. There is various error format for log output.

In the following example, the 'mysql' module is used to access the rows of a particular file. If the table connected successfully than the first row of the table will be printed to stdout using console.log. However, if an error occurs, it is logged to stderr using console.error().

**console.warn([data], [...])**

The console.warn() method is same as console.error() method. In the following example, the 'fs' module is used to open a particular file. If the file opens successfully, than console.log() prints a message. However, if an error occurs, it is logged to stderr using console.warn().

**console.dir(obj)**

Uses util.inspect on obj and prints resulting string to stdout.

**console.time(label)**

The console.time() method starts a timer and you can use it to track how long an operation takes. Each time must have and unique name (label). When you call console.timeEnd() with the same name, it prints the amount of time that has passed since a particular process started.

**console.timeEnd(label)**

The console.timeEnd() method is used to stop a timer that was previously started by calling console.time() method.

**console.trace(label)**

The console.trace() method is used to prints a stack trace to stderr of the current position.

**console.assert(expression, [message])**

The console.assert() method tests whether an expression is true. If the expression evaluates as false it will throw an AssertionError with a message.

**Module 3:- Node JS Modules**

**Functions**

JavaScript is a functional programming language, functions are fully typed objects that can be manipulated, extended, and passed around as data.

A normal function structure in JavaScript is defined as follows.

function functionName() {

// function body

// optional return;

}

All functions return a value in JavaScript.

In the absence of an explicit return statement, a function returns undefined.

function myData() {

return 123;

}

console.log(myData()); **// 123**

function myValue() {

}

console.log(myValue()); **// undefined**

Note

To declare parameters for a function in JavaScript, list them in the parentheses.

There is no checking of these parameters at runtime:

function hello(name) {

console.log(**"hello "** + name);

}

hello();

hello(**"CSS"**, **"HTML"**, **"AAA"**, 4);

The code above generates the following result.

Hello undefined

Hello css

If too few parameters are passed into a function call, the resulting variables are assigned the value undefined.

If too many are passed in, the extras are simply unused.

All functions have a predefined array in the body called arguments.

It has all the values that were passed in to the function, and we can do extra checking on the parameter list.

Functions in JavaScript do not even need to have names:

**var** x = function (a, b) {

return a + b;

}

console.log(x(10, 20));

The code above generates the following result.

30

The nameless functions are typically called anonymous functions.

Function Scope

Every time a function is called, a new variable scope is created.

Variables declared in the parent scope are available to that function.

Variables declared within the new scope are not available when the function exits.

Consider the following code:

**var** pet = **'cat'**;

function myMethod() {

**var** pet = **'dog'**;

console.log(pet);

}

myMethod();

console.log(pet);

The code above generates the following result.

Combining this scoping with anonymous functions is better way to use private variables that will disappear when the anonymous function exits.

Example 2

Here's a contrived example to compute the volume of a cone:

**var** **height** = 5;

**var** radius = 3;

**var** volume;

**// declare and immediately call anonymous function to create scope**

(function () {/\*from w w w . j av a 2 s . c o m\*/

**var** pir2 = Math.PI \* radius \* radius; **// temp var**

volume = (pir2 \* **height**) / 3;

})();

console.log(volume);

The code above generates the following result.

http://www.java2s.com/Tutorials/JavascriptImage/myResult/E/EXAMPLE_2__2BBA1C1E0F133FE3058F.PNG

**Buffer**

Node.js provides Buffer class to store raw data similar to an array of integers but corresponds to a Raw memory allocation outside the V8 heap. Buffer class is used because pure JavaScript is not nice to binary data. So, when dealing with TCP streams or the file system, it's necessary to handle octet streams. 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. It can be accessed in application without importing buffer module.

**Node.js Creating Buffers**

There are many ways to construct a Node buffer. Following are the three mostly used methods:

Create an uninitiated buffer: Following is the syntax of creating an uninitiated buffer of 10 octets:

var buf = new Buffer(10);

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

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

Create a buffer from string: 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");

**Node.js Writing to buffers**

Following is the method to write into a Node buffer:

Syntax:

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

Parameter explanation:

string: It specifies the string data to be written to buffer.

offset: It specifies the index of the buffer to start writing at. Its default value is 0.

length: It specifies the number of bytes to write. Defaults to buffer.length

encoding: Encoding to use. 'utf8' is the default encoding.

Return values from writing buffers:

This method is used to return number of octets written.

In the case of space shortage for buffer to fit the entire string, it will write a part of the string.

Let's take an example:

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

buf = new Buffer(256);

len = buf.write("Simply Easy Learning");

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

**Node.js Reading from buffers**

Following is the method to read data from a Node buffer.

Syntax:

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

Parameter explanation:

encoding: It specifies encoding to use. 'utf8' is the default encoding

start: It specifies beginning index to start reading, defaults to 0.

end: It specifies end index to end reading, defaults is complete buffer.

Return values reading from buffers:

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

Let's take an 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

**Convert Buffer to JSON**

Syntax

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

var buf = new Buffer('Simply Easy Learning');

var json = buf.toJSON(buf);

console.log(json);

**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.

**Live Demo**

var buffer1 = new Buffer('Node js');

var buffer2 = new Buffer('Simply Easy Learning');

var buffer3 = Buffer.concat([buffer1,buffer2]);

**console.log("buffer3 content: " + buffer3.toString());**

**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.

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);

}

**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

Live Demo

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.

var buffer1 = new Buffer('TutorialsPoint');

//slicing a buffer

var buffer2 = buffer1.slice(0,9);

console.log ("buffer2 content: " + buffer2.toString());

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.

Live Demo

var buffer = new Buffer('TutorialsPoint');

//length of the buffer

console.log ("buffer length: " + buffer.length);

**Module**

**What are modules in Node.js?**

As stated earlier, modules in Node js are a way of encapsulating code in a separate logical unit. There are many readymade modules available in the market which can be used within Node js.

Below are some of the popular modules which are used in a Node js application

1. **Express framework** – Express is a minimal and flexible Node js web application framework that provides a robust set of features for the web and[mobile](https://www.guru99.com/mobile-testing.html)applications.
2. **Socket.io** - Socket.IO enables real-time bidirectional event-based communication. This module is good for creation of chatting based applications.
3. **Jade**- Jade is a high-performance template engine and implemented with[JavaScript](https://www.guru99.com/interactive-javascript-tutorials.html)for node and browsers.
4. **MongoDB** - The[MongoDB](https://www.guru99.com/mongodb-tutorials.html)Node.js driver is the officially supported node.js driver for MongoDB.
5. **Restify** - restify is a lightweight framework, similar to express for building REST APIs
6. **Bluebird**- Bluebird is a fully-featured promise library with a focus on innovative features and performance

**Using modules in Node.js**

In order to use modules in a Node.js application, they first need to be installed using the Node package manager.

The below command line shows how a module "express" can be installed.

**npm install express**

**Module 4:- Node Package Manager**

Node Package Manager (NPM) is a command line tool that installs, updates or uninstalls Node.js packages in your application. It is also an online repository for open-source Node.js packages. The node community around the world creates useful modules and publishes them as packages in this repository.

It has now become a popular package manager for other open-source JavaScript frameworks like AngularJS, jQuery, Gulp, Bower etc.

Official website: [https://www.npmjs.com](https://www.npmjs.com/)

NPM is included with Node.js installation. After you install Node.js, verify NPM installation by writing the following command in terminal or command prompt.

 npm -v

If you have an older version of NPM then you can update it to the latest version using the following command.

npm install npm -g

To access NPM help, write **npm help** in the command prompt or terminal window.

npm help

NPM performs the operation in two modes: global and local. In the global mode, NPM performs operations which affect all the Node.js applications on the computer whereas in the local mode, NPM performs operations for the particular local directory which affects an application in that directory only.

## Install Package Locally

Use the following command to install any third party module in your local Node.js project folder.

npm install <package name>

For example, the following command will install ExpressJS into MyNodeProj folder.

C:\MyNodeProj> npm install expressexpress

All the modules installed using NPM are installed under **node\_modules** folder. The above command will create ExpressJS folder under node\_modules folder in the root folder of your project and install Express.js there.

## Add Dependency into package.json

Use --save at the end of the install command to add dependency entry into package.json of your application.

For example, the following command will install ExpressJS in your application and also adds dependency entry into the package.json.

:\MyNodeProj> npm install express –save

he package.json of NodejsConsoleApp project will look something like below.

package.json

 Copy

{

"name": "NodejsConsoleApp",

"version": "0.0.0",

"description": "NodejsConsoleApp",

"main": "app.js",

"author": {

"name": "Dev",

"email": ""

},

"dependencies": {

"express": "^4.13.3"

}

}

## Install Package Globally

NPM can also install packages globally so that all the node.js application on that computer can import and use the installed packages. NPM installs global packages into */<User>/local/lib/node\_modules* folder.

Apply -g in the install command to install package globally. For example, the following command will install ExpressJS globally.

C:\MyNodeProj> npm install -g express

## Update Package

To update the package installed locally in your Node.js project, navigate the command prompt or terminal window path to the project folder and write the following update command.

C:\MyNodeProj> npm update <package name>

The following command will update the existing ExpressJS module to the latest version.

C:\MyNodeProj> npm update express

## Uninstall Packages

Use the following command to remove a local package from your project.

C:\>npm uninstall <package name>

The following command will uninstall ExpressJS from the application.

C:\MyNodeProj> npm uninstall express

**Module 5:-** - **Creating Web server**

To access web pages of any web application, you need a [web server](https://en.wikipedia.org/wiki/Web_server). The web server will handle all the http requests for the web application e.g IIS is a web server for ASP.NET web applications and Apache is a web server for PHP or Java web applications.

Node.js provides capabilities to create your own web server which will handle HTTP requests asynchronously. You can use IIS or Apache to run Node.js web application but it is recommended to use Node.js web server.

## Create Node.js Web Server

Node.js makes it easy to create a simple web server that processes incoming requests asynchronously.

The following example is a simple Node.js web server contained in server.js file.

var http = require('http'); // 1 - Import Node.js core module

var server = http.createServer(function (req, res) { // 2 - creating server

//handle incomming requests here..

});

server.listen(5000); //3 - listen for any incoming requests

console.log('Node.js web server at port 5000 is running..')

In the above example, we import the http module using require() function. The http module is a core module of Node.js, so no need to install it using NPM. The next step is to call createServer() method of http and specify callback function with request and response parameter. Finally, call listen() method of server object which was returned from createServer() method with port number, to start listening to incoming requests on port 5000. You can specify any unused port here.

Run the above web server by writing node server.js command in command prompt or terminal window and it will display message as shown below.

C:\>node server.js  
Node.js web server at port 5000 is running..

This is how you create a Node.js web server using simple steps. Now, let's see how to handle HTTP request and send response in Node.js web server.

## Handle HTTP Request

The http.createServer() method includes [request](https://nodejs.org/api/http.html#http_http_incomingmessage) and [response](https://nodejs.org/api/http.html#http_class_http_serverresponse) parameters which is supplied by Node.js. The request object can be used to get information about the current HTTP request e.g., url, request header, and data. The response object can be used to send a response for a current HTTP request.

The following example demonstrates handling HTTP request and response in Node.js.

var http = require('http'); // Import Node.js core module

var server = http.createServer(function (req, res) { //create web server

if (req.url == '/') { //check the URL of the current request

// set response header

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

// set response content

res.write('<html><body><p>This is home Page.</p></body></html>');

res.end();

}

else if (req.url == "/student") {

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

res.write('<html><body><p>This is student Page.</p></body></html>');

res.end();

}

else if (req.url == "/admin") {

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

res.write('<html><body><p>This is admin Page.</p></body></html>');

res.end();

}

else

res.end('Invalid Request!');

});

server.listen(5000); //6 - listen for any incoming requests

console.log('Node.js web server at port 5000 is running..')

In the above example, req.url is used to check the url of the current request and based on that it sends the response. To send a response, first it sets the response header using writeHead() method and then writes a string as a response body using write() method. Finally, Node.js web server sends the response using end() method.

Now, run the above web server as shown below.

Node server.js  
Node.js web server at port 5000 is running..

## Sending JSON Response

The following example demonstrates how to serve JSON response from the Node.js web server.

var http = require('http');

var server = http.createServer(function (req, res) {

if (req.url == '/data') { //check the URL of the current request

res.writeHead(200, { 'Content-Type': 'application/json' });

res.write(JSON.stringify({ message: "Hello World"}));

res.end();

}

});

server.listen(5000);

console.log('Node.js web server at port 5000 is running..')

**Module 6:- File System**

Node.js includes **fs** module to access physical file system. The fs module is responsible for all the asynchronous or synchronous file I/O operations.

Let's see some of the common I/O operation examples using fs module.

## Reading File

Use fs.readFile() method to read the physical file asynchronously.

fs.readFile(fileName [,options], callback)

Parameter Description:

* filename: Full path and name of the file as a string.
* options: The options parameter can be an object or string which can include encoding and flag. The default encoding is utf8 and default flag is "r".
* callback: A function with two parameters err and fd. This will get called when readFile operation completes.

The following example demonstrates reading existing TestFile.txt asynchronously.

var fs = require('fs');

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

if (err) throw err;

console.log(data);

});

The above example reads TestFile.txt (on Windows) asynchronously and executes callback function when read operation completes. This read operation either throws an error or completes successfully. The err parameter contains error information if any. The data parameter contains the content of the specified file.

The following is a sample TextFile.txt file.

This is test file to test fs module of Node.js

Now, run the above example and see the result as shown below.

C:\>node server.js  
This is test file to test fs module of Node.js

Use fs.readFileSync() method to read file synchronously as shown below.

var fs = require('fs');

var data = fs.readFileSync('dummyfile.txt', 'utf8');

console.log(data);

## Writing File

Use fs.writeFile() method to write data to a file. If file already exists then it overwrites the existing content otherwise it creates a new file and writes data into it.

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

Parameter Description:

* filename: Full path and name of the file as a string.
* Data: The content to be written in a file.
* options: The options parameter can be an object or string which can include encoding, mode and flag. The default encoding is utf8 and default flag is "r".
* callback: A function with two parameters err and fd. This will get called when write operation completes.

The following example creates a new file called test.txt and writes "Hello World" into it asynchronously.

var fs = require('fs');

fs.writeFile('test.txt', 'Hello World!', function (err) {

if (err)

console.log(err);

else

console.log('Write operation complete.');

});

In the same way, use fs.appendFile() method to append the content to an existing file.

var fs = require('fs');

fs.appendFile('test.txt', 'Hello World!', function (err) {

if (err)

console.log(err);

else

console.log('Append operation complete.');

});

## Open File

Alternatively, you can open a file for reading or writing using fs.open() method.

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

Parameter Description:

* path: Full path with name of the file as a string.
* Flag: The flag to perform operation
* Mode: The mode for read, write or readwrite. Defaults to 0666 readwrite.
* callback: A function with two parameters err and fd. This will get called when file open operation completes.

### Flags

The following table lists all the flags which can be used in read/write operation.

| 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, telling 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 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 path exists. |
| a | Open file for appending. The file is created if it does not exist. |
| ax | Like 'a' but fails if path exists. |
| a+ | Open file for reading and appending. The file is created if it does not exist. |
| ax+ | Like 'a+' but fails if path exists. |

The following example opens an existing file and reads its content.

var fs = require('fs');

fs.open('TestFile.txt', 'r', function (err, fd) {

if (err) {

return console.error(err);

}

var buffr = new Buffer(1024);

fs.read(fd, buffr, 0, buffr.length, 0, function (err, bytes) {

if (err) throw err;

// Print only read bytes to avoid junk.

if (bytes > 0) {

console.log(buffr.slice(0, bytes).toString());

}

// Close the opened file.

fs.close(fd, function (err) {

if (err) throw err;

});

});

});

## Delete File

Use fs.unlink() method to delete an existing file.

fs.unlink(path, callback);

The following example deletes an existing file.

var fs = require('fs');

fs.unlink('test.txt', function () {

console.log('write operation complete.');

});

Important method of fs module

| Method | Description |
| --- | --- |
| fs.readFile(fileName [,options], callback) | Reads existing file. |
| fs.writeFile(filename, data[, options], callback) | Writes to the file. If file exists then overwrite the content otherwise creates new file. |
| fs.open(path, flags[, mode], callback) | Opens file for reading or writing. |
| fs.rename(oldPath, newPath, callback) | Renames an existing file. |
| fs.chown(path, uid, gid, callback) | Asynchronous chown. |
| fs.stat(path, callback) | Returns fs.stat object which includes important file statistics. |
| fs.link(srcpath, dstpath, callback) | Links file asynchronously. |
| fs.symlink(destination, path[, type], callback) | Symlink asynchronously. |
| fs.rmdir(path, callback) | Renames an existing directory. |
| fs.mkdir(path[, mode], callback) | Creates a new directory. |
| fs.readdir(path, callback) | Reads the content of the specified directory. |
| fs.utimes(path, atime, mtime, callback) | Changes the timestamp of the file. |
| fs.exists(path, callback) | Determines whether the specified file exists or not. |
| fs.access(path[, mode], callback) | Tests a user's permissions for the specified file. |
| fs.appendFile(file, data[, options], callback) | Appends new content to the existing file. |

**Module 7:- Debugging Node JS Application**

Debug Node.js Application

In this section, you will learn ways to debug Node.js application.You can debug Node.js application using various tools including following:

Core Node.js debugger

Node Inspector

Built-in debugger in IDEs

**Core Node.js Debugger**

Node.js provides built-in non-graphic debugging tool that can be used on all platforms. It provides different commands for debugging Node.js application.

Consider the following simple Node.js application contained in app.js file.

var fs = require('fs');

fs.readFile('test.txt', 'utf8', function (err, data) {

debugger;

if (err) throw err;

console.log(data);

});

Write debugger in your JavaScript code where you want debugger to stop. For example, we want to check the "data" parameter in the above example. So, write debugger; inside callback function as above.

Now, to debug the above application, run the following command.

node debug app.js

The above command starts the debugger and stops at the first line as shown below.As you can see in the above figure, > symbol indicates the current debugging statement.

Use next to move on the next statement.In the above figure, next command will set the debugger on the next line. The > is now pointing to next statement.Use cont to stop the execution at next "debugger", if any.

Use watch('expression') command to add the variable or expression whose value you want to check. For example, to check the value of data variable in the above example, add data into watch expression as shown below.Now, write watchers command to check the value of all the variables added into watch().

The following table lists important debugging commands:

Command Description

next Stop at the next statement.

cont Continue execute and stop at the debugger statement if any.

step Step in function.

out Step out of function.

watch Add the expression or variable into watch.

Watcher See the value of all expressions and variables added into watch.

Pause Pause running code.

**Module 8:- Events**

Node.js EventEmitter

Node.js allows us to create and handle custom events easily by using events module.

Event module includes EventEmitter class which can be used to raise and handle custom events.

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.

**Node.js Events**

In Node.js applications, Events and Callbacks concepts are used to provide concurrency. As Node.js

Applications are single threaded and every API of Node js are asynchronous. So it uses async

Function to maintain the concurrency. Node uses observer pattern. Node thread keeps an event

Loop and after the completion of any task, it fires the corresponding event which signals the

Event listener function to get executed.

**Event Driven Programming**

Node.js uses event driven programming. It means as soon as Node starts its server, it simply

Initiates its variables, declares functions and then simply waits for event to occur. It is the one

Of the reason why Node.js is pretty fast compared to other similar technologies.

There is a main loop in the event driven application that listens for events, and then triggers a

callback function when one of those events is detected.

EventEmitter class to bind event and event listener:

**// Import events module**

var events = require('events');

// Create an eventEmitter object

var eventEmitter = new events.EventEmitter();

**// Import events module**

var events = require('events');

// Create an eventEmitter object

var eventEmitter = new events.EventEmitter();

**To bind event handler with an event:**

// Bind event and even handler as follows

eventEmitter.on('eventName', eventHandler);

**To fire an event:**

**// Fire an event**

eventEmitter.emit('eventName');

**EventEmitter Class**

As we have seen in the previous section, EventEmitter class lies in the events module.

It is accessible via the following code −The following example demonstrates EventEmitter class for raising and handling a custom event.

// get the reference of EventEmitter class of events module

var events = require('events');

//create an object of EventEmitter class by using above reference

var em = new events.EventEmitter();

//Subscribe for FirstEvent

em.on('FirstEvent', function (data) {

console.log('First subscriber: ' + data);

});

// Raising FirstEvent

em.emit('FirstEvent', 'This is my first Node.js event emitter example.');

In the above example, we first import the 'events' module and then create an object of

EventEmitter class. We then specify event handler function using on() function. The on()

method requires name of the event to handle and callback function which is

called when an event is raised.

The emit() function raises the specified event. First parameter is name of the event as

a string and then arguments. An event can be emitted with zero or more arguments.

You can specify any name for a custom event in the emit() function.

You can also use addListener() methods to subscribe for an event as shown below.

var emitter = require('events').EventEmitter;

var em = new emitter();

//Subscribe FirstEvent

em.addListener('FirstEvent', function (data) {

console.log('First subscriber: ' + data);

});

//Subscribe SecondEvent

em.on('SecondEvent', function (data) {

console.log('First subscriber: ' + data);

});

// Raising FirstEvent

em.emit('FirstEvent', 'This is my first Node.js event emitter example.');

// Raising SecondEvent

em.emit('SecondEvent', 'This is my second Node.js event emitter example.');

The EventEmitter Object

You can assign event handlers to your own events with the EventEmitter object.

In the example below we have created a function that will be executed when a

"scream" event is fired.

To fire an event, use the emit() method.

var events = require('events');

var eventEmitter = new events.EventEmitter();

//Create an event handler:

var myEventHandler = function () {

console.log('Welcome to Node JS');

}

//Assign the event handler to an event:

eventEmitter.on('node', myEventHandler);

//Fire the 'node' event:

eventEmitter.emit('node');

The following table lists all the important methods of EventEmitter class.

**EventEmitter Methods Description**

emitter.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.

emitter.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. It can also be called as an alias of emitter.addListener()

emitter.once(event, listener) Adds a one time listener for the event. This listener is invoked only the

next time the event is fired, after which it is removed.

emitter.removeListener(event, listener) Removes a listener from the listener array for the specified event.

Caution: changes array indices in the listener array behind the listener.

emitter.removeAllListeners([event]) Removes all listeners, or those of the specified event.

emitter.setMaxListeners(n) By default EventEmitters will print a warning if more than 10 listeners are added

for a particular event.

emitter.getMaxListeners() Returns the current maximum listener value for the emitter which is either set by

emitter.setMaxListeners(n) or defaults to EventEmitter.defaultMaxListeners.

emitter.listeners(event) Returns a copy of the array of listeners for the specified event.

emitter.emit(event[, arg1][, arg2][, ...]) Raise the specified events with the supplied arguments.

emitter.listenerCount(type) Returns the number of listeners listening to the type of event.

Common Patterns for EventEmitters

There are two common patterns that can be used to raise and bind an event using

EventEmitter class in Node.js.

Return EventEmitter from a function

Extend the EventEmitter class

Return EventEmitter from a function

In this pattern, a constructor function returns an EventEmitter object, which was used to emit

events inside a function. This EventEmitter object can be used to subscribe for the events.

Consider the following example.

var emitter = require('events').EventEmitter;

function LoopProcessor(num) {

var e = new emitter();

setTimeout(function ()

for (var i = 1; i <= num; i++) {

e.emit('BeforeProcess', i);

console.log('Processing number:' + i);

e.emit('AfterProcess', i);

}

}

, 2000)

return e;

}

var lp = LoopProcessor(3);

lp.on('BeforeProcess', function (data) {

console.log('About to start the process for ' + data);

});

lp.on('AfterProcess', function (data) {

console.log('Completed processing ' + data);

});

In the above LoopProcessor() function, first we create an object of EventEmitter class

And then use it to emit 'BeforeProcess' and 'AfterProcess' events. Finally, we return an object of EventEmitter from the function. So now, we can use the return value of LoopProcessor function to bind these events using on() or addListener() function.

**Extend EventEmitter Class**

In this pattern, we can extend the constructor function from EventEmitter class to emit the events.

var emitter = require('events').EventEmitter;

var util = require('util');

function LoopProcessor(num) {

var me = this;

setTimeout(function () {

for (var i = 1; i <= num; i++) {

me.emit('BeforeProcess', i);

console.log('Processing number:' + i);

me.emit('AfterProcess', i);

}

}

, 2000)

return this;

}

util.inherits(LoopProcessor, emitter)

var lp = new LoopProcessor(3);

lp.on('BeforeProcess', function (data) {

console.log('About to start the process for ' + data);

});

lp.on('AfterProcess', function (data) {

console.log('Completed processing ' + data);

});

In the above example, we have extended LoopProcessor constructor function with EventEmitter

Class using util.inherits() method of utility module. So, you can use EventEmitter's methods

With LoopProcessor object to handle its own events.

In this way, you can use EventEmitter class to raise and handle custom events in Node.js.

**Module 9:- Working with asynchronous programming**

Asynchronous I/O is a form of input/output processing that permits other processing to continue before the transmission has finished.In the following example, I will show you a simple file reading process in Node.js - both in a synchronous and asynchronous way, with the intention of show you what can be achieved by avoiding blocking your applications.

Let's start with a simple example - reading a file using Node.js in a synchronous way:

const fs = require('fs')

let content

try {

content = fs.readFileSync('file.md', 'utf-8')

} catch (ex) {

console.log(ex)

}

console.log(content)

What did just happen here? We tried to read a file using the synchronous interface of the fs module. It works as expected - the content variable will contain the content of file.md. The problem with this approach is that Node.js will be blocked until the operation is finished - meaning it can do absolutely nothing while the file is being read.

Let's see how we can fix it!

Asynchronous programming - as we know now in JavaScript - can only be achieved with functions being first-class citizens of the language: they can be passed around like any other variables to other functions. Functions that can take other functions as arguments are called higher-order functions.

One of the easiest example for higher order functions:

const numbers = [2,4,1,5,4]

function isBiggerThanTwo (num) {

return num > 2

}

numbers.filter(isBiggerThanTwo)

In the example above we pass in a function to the filter function. This way we can define the filtering logic.

This is how callbacks were born: if you pass a function to another function as a parameter, you can call it within the function when you are finished with your job. No need to return values, only calling another function with the values.

These so-called error-first callbacks are in the heart of Node.js itself - the core modules are using it as well as most of the modules found on NPM.

const fs = require('fs')

fs.readFile('file.md', 'utf-8', function (err, content) {

if (err) {

return console.log(err)

}

console.log(content)

})

Things to notice here:

error-handling: instead of a try-catch block you have to check for errors in the callback

no return value: async functions don't return values, but values will be passed to the callbacks

**Callback functions**

A JavaScript function is a block of code that will be executed when you call it Because JavaScript functions are first-class objects, you can pass functions to other functions as variables The method of passing in functions as parameters to other functions to use them inside is used in JavaScript libraries almost everywhere

A JavaScript Callback Function is a function that is passed as a parameter to another JavaScript function, and the callback function is run inside of the function it was passed intoJavaScript Callback Functions can be used synchronously or asynchronously

function functionOne(x) { return x; };

function functionTwo(var1) {

// some code

}

functionTwo(functionOne);

**Working with Promises**

A promise is an object that may produce a single value some time in the future: either a resolved value, or a reason that it’s not resolved (e.g., a network error occurred). A promise may be in one of 3 possible states: fulfilled, rejected, or pending. Promise users can attach callbacks to handle the fulfilled value or the reason for rejection.

let promise = new Promise(function(resolve, reject) {

// executor (the producing code, "singer")

});

The function passed to new Promise is called the executor. When new Promise is created, the executor runs automatically. It contains the producing code which should eventually produce the result. In terms of the analogy above: the executor is the “singer”.

Its arguments resolve and reject are callbacks provided by JavaScript itself. Our code is only inside the executor.When the executor obtains the result, be it soon or late, doesn’t matter, it should call one of these callbacks:

resolve(value) — if the job finished successfully, with result value.

reject(error) — if an error occurred, error is the error object.

So to summarize: the executor runs automatically and attempts to perform a job. When it is finished with the attempt it calls resolve if it was successful or reject if there was an error.

The promise object returned by the new Promise constructor has these internal properties:

state — initially "pending", then changes to either "fulfilled" when resolve is called or "rejected" when reject is called.

result — initially undefined, then changes to value when resolve(value) called or error when reject(error) is called.

So the executor eventually moves promise to one of these states:

**Module 10:- Express.JS**

What is Express.js?

Express.js is a Node js web application server framework, which is specifically designed for building single-page, multi-page, and hybrid web applications.

It has become the standard server framework for node.js. Express is the backend part of something known as the MEAN stack.

The MEAN is a free and open-source JavaScript software stack for building dynamic web sites and web applications which has the following components;

1) MongoDB - The standard NoSQL database

2) Express.js - The default web applications framework

3) Angular.js - The JavaScript MVC framework used for web applications

4) Node.js - Framework used for scalable server-side and networking applications.

The Express.js framework makes it very easy to develop an application which can be used to

Handle multiple types of requests like the GET, PUT, and POST and DELETE requests.

Installing and using Express

Express gets installed via the Node Package Manager.

This can be done by executing the following line in the

command line

npm install express

The above command requests the Node package manager to download the required express modules and

install them accordingly.

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 setting 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

Let's use our newly installed Express framework and create a simple "Hello World" application.

Our application is going to create a simple server module which will listen on port number 3000. In our example, if a request is made through the browser on this port number, then server

application will send a 'Hello' World' response to the client.

var express=require('express');

var app=express();

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

{

res.send('Hello World!');

});

var server=app.listen(3000,function() {});

Code Explanation:

In our first line of code, we are using the require function to include the "express module."Before we can start using the express module, we need to make an object of it.Here we are creating a callback function. This function will be called whenever anybody browses to the root of our web application which is http://localhost:3000 .

The callback function will be used to send the string 'Hello World' to the web page.In the callback function, we are sending the string "Hello World" back to the client. The 'res' parameter is used to send content back to the web page. This 'res' parameter is something that is provided by the 'request' module to enable one to send content back to the web page.

We are then using the listen to function to make our server application listen to client requests on port no 3000. You can specify any available port over here.

If the command is executed successfully, the following Output will be shown when you run your code in the browser.

From the output,

You can clearly see that we if browse to the URL of localhost on port 3000, you will see the string 'Hello World' displayed on the page.Because in our code we have mentioned specifically for the server to listen on port no 3000,we are able to view the output when browsing to this URL.

Hello world Example

Following is a very basic Express app which starts a server and listens on

port 8081 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)

})

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.

What are Routes?

Routing determine the way in which an application responds to a client request to a particular endpoint.

For example, a client can make a GET, POST, PUT or DELETE http request for various URL such as the ones shown below;

http://localhost:3000/Books

http://localhost:3000/Students

In the above example,

If a GET request is made for the first URL, then the response should ideally be a list of books.If the GET request is made for the second URL, then the response should ideally be a list of Students.So based on the URL which is accessed, a different functionality on the webserver will be invoked, and accordingly, the response will be sent to the client. This is the concept of routing.

Each route can have one or more handler functions, which are executed when the route is matched.

The general syntax for a route is shown below

app.METHOD(PATH, HANDLER)

Wherein,

1) app is an instance of the express module

2) METHOD is an HTTP request method (GET, POST, PUT or DELETE)

3) PATH is a path on the server.

4) HANDLER is the function executed when the route is matched.

Let's look at an example of how we can implement routes in the express.

Our example will create 3 routes as

A /Node route which will display the string "Tutorial on Node" if this route is accessed

A /Angular route which will display the string "Tutorial on Angular" if this route is accessed

A default route / which will display the string "Welcome to Node JS Tutorials."

var express = require('express');

var app = express();

app.route('/Node').get(function(req,res)

{

res.send("Tutorial on Node");

});

app.route('/Angular').get(function(req,res)

{

res.send("Tutorial on Angular");

});

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

res.send('Welcome to Node Tutorials');

}));

Code Explanation:

Here we are defining a route if the URL http://localhost:3000/Node is selected in the browser. To the route,we are attaching a callback function which will be called when we browse to the Node URL.The function has 2 parameters.The main parameter we will be using is the 'res' parameter, which can be used to send information back to the client.The 'req' parameter has information about the request being made. Sometimes additional parameters could be sent as part of the request being made, and hence the 'req' parameter can be used to find the additional parameters being sent.

We are using the send function to send the string "Tutorial on Node" back to the client if the Node route is chosen.

Here we are defining a route if the URL http://localhost:3000/Angular is selected in the browser.

To the route, we are attaching a callback function which will be called when we browse to the Angular URL.

We are using the send function to send the string "Tutorial on Angular" back to the client if the

Angular route is chosen.

This is the default route which is chosen when one browses to the route of the application –

http://localhost:3000. When the default route is chosen, the message "Welcome to Node Tutorials" will be sent to the client.

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');

})

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)

})

Web Application

In this section, you will learn how to create a web application using Express.js.Express.js provides an easy way to create web server and render HTML pages for different HTTP requests by configuring routes for your application.

Web Server

First of all, import the Express.js module and create the web server as shown below.

app.js: Express.js

var express = require('express');

var app = express();

// define routes here..

var server = app.listen(5000, function () {

console.log('Node server is running..');

});

In the above example, we imported Express.js module using require() function. The express module returns a function. This function returns an object which can be used to configure Express application (app in the above example).

The app object includes methods for routing HTTP requests, configuring middleware, rendering HTML views and registering a template engine.

The app.listen() function creates the Node.js web server at the specified host and port.

It is identical to Node's http.Server.listen() method.

Run the above example using node app.js command and point your browser to http://localhost:5000.

It will display Cannot GET / because we have not configured any routes yet.

Configure Routes

Use app object to define different routes of your application. The app object includes get(), post(),

put() and delete() methods to define routes for HTTP GET, POST, PUT and DELETE requests respectively.

The following example demonstrates configuring routes for HTTP requests.

var express = require('express');

var app = express();

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

res.send('<html><body><h1>Hello World</h1></body></html>');

});

app.post('/submit-data', function (req, res) {

res.send('POST Request');

});

app.put('/update-data', function (req, res) {

res.send('PUT Request');

});

app.delete('/delete-data', function (req, res) {

res.send('DELETE Request');

});

var server = app.listen(5000, function () {

console.log('Node server is running..');

});

In the above example, app.get(), app.post(), app.put() and app.delete() methods define routes for HTTP GET, POST, PUT, DELETE respectively. The first parameter is a path of a route which will start after base URL. The callback function includes request and response object which will be executed on each request.

Run the above example using node server.js command, and point your browser to http://localhost:5000 and you will see the following result.

Handle POST Request

Here, you will learn how to handle HTTP POST request and get data from the submitted form. First, create Index.html file in the root folder of your application and write the following HTML code in it.

<!DOCTYPE html>

<html xmlns="http://www.w3.org/1999/xhtml">

<head>

<meta charset="utf-8" />

<title></title>

</head>

<body>

<form action="/submit-student-data" method="post">

First Name: <input name="firstName" type="text" /> <br />

Last Name: <input name="lastName" type="text" /> <br />

<input type="submit" />

</form>

</body>

</html>

Body Parser

To handle HTTP POST request in Express.js version 4 and above, you need to install middleware module

called body-parser. The middleware was a part of Express.js earlier but now you have to install it separately.This body-parser module parses the JSON, buffer, string and url encoded data submitted using HTTP POST request.

Install body-parser using NPM as shown below.

npm install body-parser --save

Now, import body-parser and get the POST request data as shown below.

var express = require('express');

var app = express();

var bodyParser = require("body-parser");

app.use(bodyParser.urlencoded({ extended: false }));

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

res.sendFile('index.html');

});

app.post('/submit-student-data', function (req, res) {

var name = req.body.firstName + ' ' + req.body.lastName;

res.send(name + ' Submitted Successfully!');

});

var server = app.listen(5000, function () {

console.log('Node server is running..');

});

0In the above example, POST data can be accessed using req.body. The req.body is an object that includes properties for each submitted form. Index.html contains firstName and lastName input types, so you can access it using req.body.firstName and req.body.lastName.

Now, run the above example using node server.js command, point your browser to http://localhost:5000 and see the following result.

Fill the First Name and Last Name in the above example and click on submit. For example, enter "James" in First Name textbox and "Bond" in Last Name textbox and click the submit button.

**Module 11:- Serving Static Resources**

Serving Static Resources in Node.js

In this section, you will learn how to serve static resources like images, css, JavaScript or other static files using Express.js and node-static module. It is easy to serve static files using built-in middleware in Express.js called express.static. Using express.static() method, you can server static resources directly by specifying the folder name where you have stored your static resources.

The following example serves static resources from the public folder under the root folder of your application.

var express = require('Express');

var path = require('path');

var app = express();

// to set simple path

app.use(express.static(\_\_dirname+"/public"));

//app.use(express.static(\_\_dirname+"/images"));

// To set virtual path prefix

app.use("/myimages",express.static(path.join(\_\_dirname, 'images')));

app.use(express.static(\_\_dirname+"/styles"));

console.log(\_\_dirname);

app.get("/", function(req,res){

res.send('mydata is called');

})

app.get("/getmethod", function(req,res){

res.send('mydata is called');

})

app.listen(3000);

**Module 12:- Database connectivity**

Node.js can be used in database applications.One of the most popular databases is MySQL.

MySQL Database

To be able to experiment with the code examples, you should have MySQL installed on your computer.You can download a free MySQL database at https://www.mysql.com/downloads/.

Install MySQL Driver

Once you have MySQL up and running on your computer, you can access it by using Node.js.

To access a MySQL database with Node.js, you need a MySQL driver. This tutorial will use the "mysql" module, downloaded from NPM. To download and install the "mysql" module, open the Command Terminal and execute the following:

C:\Users\Your Name>npm install mysql

Now you have downloaded and installed a mysql database driver.Node.js can use this module to manipulate the MySQL database:

var mysql = require('mysql');

Create Connection

Start by creating a connection to the database. Use the username and password from your MySQL database.

demo\_db\_connection.js

var mysql = require('mysql');

var con = mysql.createConnection({

host: "localhost",

user: "yourusername",

password: "yourpassword"

});

con.connect(function(err) {

if (err) throw err;

console.log("Connected!");

});

Creating a Database

To create a database in MySQL, use the "CREATE DATABASE" statement:

Example

Create a database named "mydb":

var mysql = require('mysql');

var con = mysql.createConnection({

host: "localhost",

user: "yourusername",

password: "yourpassword"

});

con.connect(function(err) {

if (err) throw err;

console.log("Connected!");

con.query("CREATE DATABASE mydb", function (err, result) {

if (err) throw err;

console.log("Database created");

});

});

Creating a Table

To create a table in MySQL, use the "CREATE TABLE" statement. Make sure you define the name of the database when you create the connection:

Example

Create a table named "customers":

var mysql = require('mysql');

var con = mysql.createConnection({

host: "localhost",

user: "yourusername",

password: "yourpassword",

database: "mydb"

});

con.connect(function(err) {

if (err) throw err;

console.log("Connected!");

var sql = "CREATE TABLE customers (name VARCHAR(255), address VARCHAR(255))";

con.query(sql, function (err, result) {

if (err) throw err;

console.log("Table created");

});

});

Insert Into Table

To fill a table in MySQL, use the "INSERT INTO" statement.

Example

Insert a record in the "customers" table:

var mysql = require('mysql');

var con = mysql.createConnection({

host: "localhost",

user: "yourusername",

password: "yourpassword",

database: "mydb"

});

con.connect(function(err) {

if (err) throw err;

console.log("Connected!");

var sql = "INSERT INTO customers (name, address) VALUES ('Company Inc', 'Highway 37')";

con.query(sql, function (err, result) {

if (err) throw err;

console.log("1 record inserted");

});

});

Node.js MongoDB

Node.js can be used in database applications. One of the most popular NoSQL database is MongoDB.

MongoDB

To be able to experiment with the code examples, you will need access to a MongoDB database.

You can download a free MongoDB database at <https://www.mongodb.com>. Or get started right away with a MongoDB cloud service at https://www.mongodb.com/cloud/atlas.

Install MongoDB Driver Let us try to access a MongoDB database with Node.js.

To download and install the official MongoDB driver, open the Command Terminal and execute the following: Download and install mongodb package:

C:\Users\Your Name>npm install mongodb

Now you have downloaded and installed a mongodb database driver. Node.js can use this module to manipulate MongoDB databases:

var mongo = require('mongodb');

Creating a Database

To create a database in MongoDB, start by creating a MongoClient object, then specify a connection URL with the correct ip address and the name of the database you want to create.

MongoDB will create the database if it does not exist, and make a connection to it.

Example

Create a database called "mydb":

var MongoClient = require('mongodb').MongoClient;

var url = "mongodb://localhost:27017/mydb";

MongoClient.connect(url, function(err, db) {

if (err) throw err;

console.log("Database created!");

db.close();

});

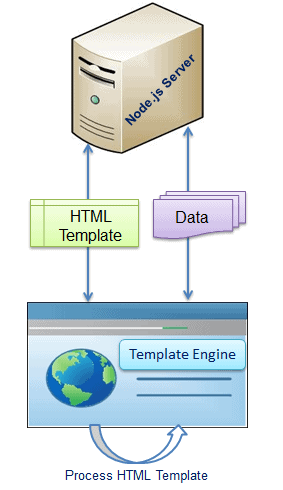
Save the code above in a file called "demo\_create\_mongo\_db.js" and run the file:

**Module 13:- Template Engines**

# Template Engines for Node.js

Template engine helps us to create an HTML template with minimal code. Also, it can inject data into HTML template at client side and produce the final HTML.

The following figure illustrates how template engine works in Node.js.



As per the above figure, client-side browser loads HTML template, JSON/XML data and template engine library from the server. Template engine produces the final HTML using template and data in client's browser. However, some HTML templates process data and generate final HTML page at server side also.

There are many template engines available for Node.js. Each template engine uses a different language to define HTML template and inject data into it.

The following is a list of important (but not limited) template engines for Node.js

* [Jade](https://github.com/jadejs/jade)
* [Vash](https://github.com/kirbysayshi/vash)
* [EJS](https://github.com/tj/ejs)
* [Mustache](https://github.com/janl/mustache.js)
* [Dust.js](https://github.com/linkedin/dustjs)
* [Nunjucks](https://github.com/mozilla/nunjucks)
* [Handlebars](https://github.com/wycats/handlebars.js)
* [atpl](https://github.com/soywiz/atpl.js)
* [haml](https://github.com/tj/haml.js)

### Advantages of Template engine in Node.js

1. Improves developer's productivity.
2. Improves readability and maintainability.
3. Faster performance.
4. Maximizes client side processing.
5. Single template for multiple pages.
6. Templates can be accessed from CDN (Content Delivery Network).

**Jade Template Engine**

Jade is a template engine for Node.js. Jade syntax is easy to learn. It uses whitespace and indentation as a part of the syntax.

Install jade into your project using NPM as below.

npm install jade

By default Express.js searches all the views in the views folder under the root folder,which can be set to another folder using views property in express e.g. app.set('views','MyViews').

<!DOCTYPE html>

<html>

<head>

<title>Jade Page</title>

</head>

<body>

<h1>This page is produced by Jade engine</h1>

<p>some paragraph here..</p>

</body>

</html>

Note: Please be careful while giving spaces and indentation in Jade. A small mistake can change the output.

**Vash Template Engine**

Vash is a template view engine that uses Razor Syntax. So, this template engine will look familiar to people who have experience in ASP.Net MVC.

Install Vash using NPM as below.

npm install vash --save

<!DOCTYPE html>

<html>

<head>

<title>@model.title</title>

</head>

<body>

<p>@model.content</p>

</body>

</html>

In the above vash template, the model object is @model and properties are title and content.

@model in razor syntax represents an object which will be supplied from the server while

Rendering this template. So, the above template expects an object with two properties: title and content.Now, create server.js and write following express.js to render the above vash template.

var express = require('express');

var app = express();

app.set("view engine","vash")

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

res.render('index', { title: 'Vash Template Demo',

content:'This is dummy paragraph.'});

});

var server = app.listen(5000, function () {

console.log('Node server is running..');

});

As you can see, we have specified an object with title and content properties in the res.render() Method while rendering index.vash. Now, run the above application using node server.js command and point your browser to http://localhost:5000 and you will get the following result.