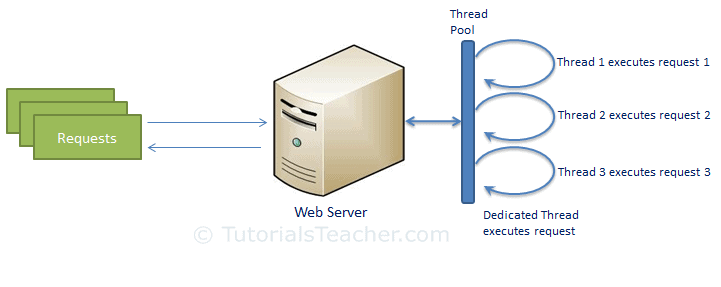
UNIT-3

Node.js Process Model

In this section, we will learn about the Node.js process model and understand why we should use Node.js.

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.

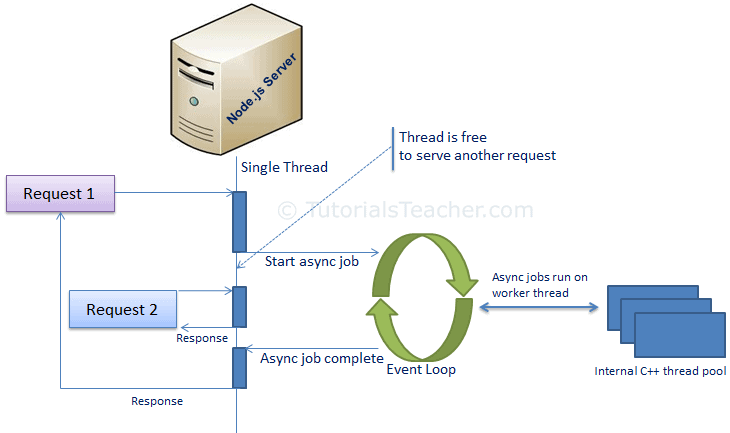
[](https://www.tutorialsteacher.com/Content/images/nodejs/traditional-web-server-model.png)Traditional Web Server Model

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.

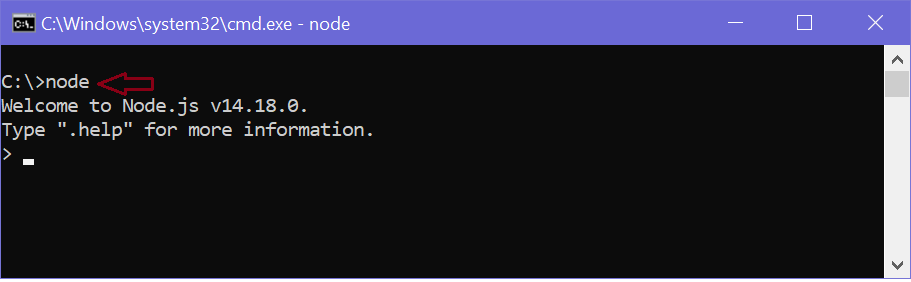
[](https://www.tutorialsteacher.com/Content/images/nodejs/nodejs-process-model.png)Node.js Process Model

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.

Node.js Console/REPL

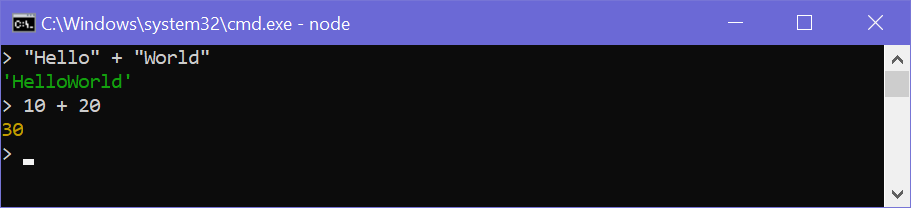
Node.js comes with virtual environment called REPL (aka Node shell). REPL stands for Read-Eval-Print-Loop. It is a quick and easy way to test simple Node.js/JavaScript code.

To launch the REPL (Node shell), open command prompt (in Windows) or terminal (in Mac or UNIX/Linux) and type *node* as shown below. It will change the prompt to > in Windows and MAC.

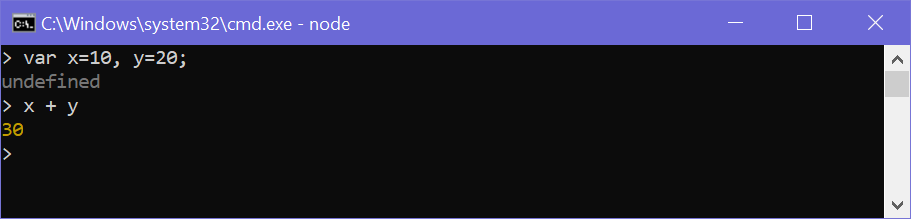
[](https://www.tutorialsteacher.com/Content/images/nodejs/node-repl.png)Launch Node.js REPL

You can now test pretty much any Node.js/JavaScript expression in REPL. 10 + 20 will display 30 immediately in new line.

The + operator also concatenates strings as in browser's JavaScript.

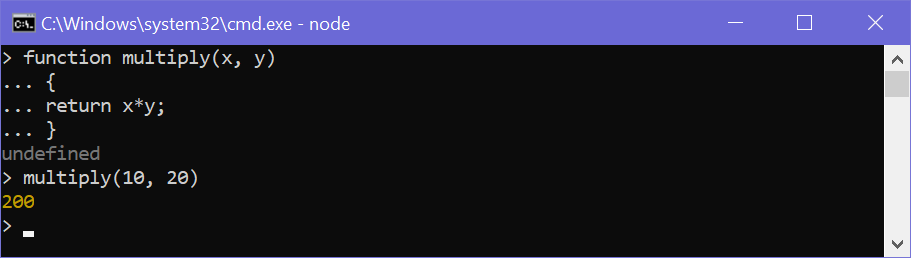
[](https://www.tutorialsteacher.com/Content/images/nodejs/node-repl-examples.png)Node.js Examples

You can also define variables and perform some operation on them.

[](https://www.tutorialsteacher.com/Content/images/nodejs/node-variables.png)Define Variables on REPL

If you need to write multi line JavaScript expression or function then just press **Enter** whenever you want to write something in the next line as a continuation of your code. The REPL terminal will display three dots (...), it means you can continue on next line. Write .break to get out of continuity mode.

For example, you can define a function and execute it as shown below.

[](https://www.tutorialsteacher.com/Content/images/nodejs/nodejs-example2.png)Node.js Example in REPL

You can execute an external JavaScript file by executing the node fileName command. For example, the following runs mynodejs-app.js on the command prompt/terminal and displays the result.

mynodejs-app.js

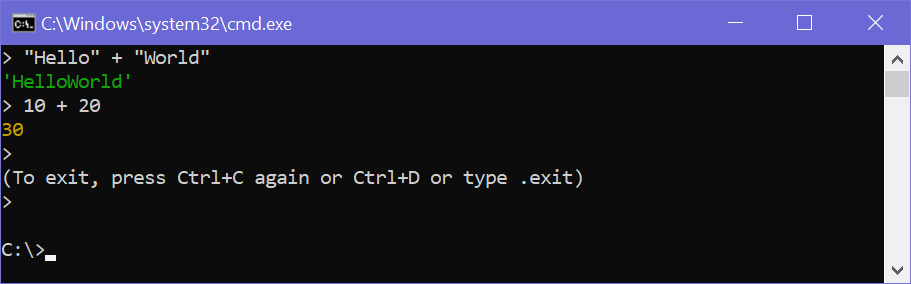
 Copy

console.log("Hello World");

Now, you can execute mynodejs-app from command prompt as shown below.

[](https://www.tutorialsteacher.com/Content/images/nodejs/run-nodejs-external-file.png)Run External JavaScript file

To exit from the REPL terminal, press Ctrl + C twice or write .exit and press Enter.

[](https://www.tutorialsteacher.com/Content/images/nodejs/quit-repl.png)Quit from REPL

Thus, you can execute any Node.js/JavaScript code in the node shell (REPL). This will give you a result which is similar to the one you will get in the console of Google Chrome browser.

 Note:

ECMAScript implementation in Node.js and browsers is slightly different. For example, {}+{} is '[object Object][object Object]' in Node.js REPL, whereas the same code is NaN in the Chrome console because of the automatic semicolon insertion feature. However, mostly Node.js REPL and the Chrome/Firefox consoles are similar.

The following table lists important REPL commands.

| REPL Command | Description |
| --- | --- |
| .help | Display help on all the commands |
| tab Keys | Display the list of all commands. |
| Up/Down Keys | See previous commands applied in REPL. |
| .save filename | Save current Node REPL session to a file. |
| .load filename | Load the specified file in the current Node REPL session. |
| ctrl + c | Terminate the current command. |
| ctrl + c (twice) | Exit from the REPL. |
| ctrl + d | Exit from the REPL. |
| .break | Exit from multiline expression. |
| .clear | Exit from multiline expression. |

Node.js Basics

Node.js supports JavaScript. So, JavaScript syntax on Node.js is similar to the browser's JavaScript syntax.

Visit [JavaScript](https://www.tutorialsteacher.com/javascript) section to learn about JavaScript syntax in detail.

Primitive Types

Node.js includes following primitive types:

* String
* Number
* Boolean
* Undefined
* Null
* RegExp

Everything else is an object in Node.js.

Loose Typing

JavaScript in Node.js supports loose typing like the browser's JavaScript. Use var keyword to declare a variable of any type.

Object Literal

Object literal syntax is same as browser's JavaScript.

Example: Object

 Copy

var obj = {

authorName: 'Ryan Dahl',

language: 'Node.js'

}

Functions

Functions are first class citizens in Node's JavaScript, similar to the browser's JavaScript. A function can have attributes and properties also. It can be treated like a class in JavaScript.

Example: Function

 Copy

function Display(x) {

console.log(x);

}

Display(100);

Buffer

Node.js includes an additional data type called Buffer (not available in browser's JavaScript). Buffer is mainly used to store binary data, while reading from a file or receiving packets over the network.

process object

Each Node.js script runs in a process. It includes **process** object to get all the information about the current process of Node.js application.

The following example shows how to get process information in REPL using **process** object.

> process.execPath  
'C:\\Program Files\\nodejs\\node.exe'  
> process.pid  
1652  
> process.cwd()  
'C:\\'

Defaults to local

Node's JavaScript is different from browser's JavaScript when it comes to global scope. In the browser's JavaScript, variables declared without var keyword become global. In Node.js, everything becomes local by default.

Access Global Scope

In a browser, global scope is the window object. In Node.js, **global** object represents the global scope.

To add something in global scope, you need to export it using export or module.export. The same way, import modules/object using require() function to access it from the global scope.

For example, to export an object in Node.js, use exports.name = object.

Example:

 Copy

exports.log = {

console: function(msg) {

console.log(msg);

},

file: function(msg) {

// log to file here

}

}

Now, you can import log object using require() function and use it anywhere in your Node.js project.

Learn about modules in detail in the next section.

# Node.js Module

Module in Node.js is a simple or complex functionality organized in single or multiple JavaScript files which can be reused throughout the Node.js application.

Each module in Node.js has its own context, so it cannot interfere with other modules or pollute global scope. Also, each module can be placed in a separate .js file under a separate folder.

Node.js implements [CommonJS modules standard](http://requirejs.org/docs/commonjs.html" \t "_blank). CommonJS is a group of volunteers who define JavaScript standards for web server, desktop, and console application.

## Node.js Module Types

Node.js includes three types of modules:

1. Core Modules
2. Local Modules
3. Third Party Modules

## Node.js Core Modules

Node.js is a light weight framework. The core modules include bare minimum functionalities of Node.js. These core modules are compiled into its binary distribution and load automatically when Node.js process starts. However, you need to import the core module first in order to use it in your application.

The following table lists some of the important core modules in Node.js.

| Core Module | Description |
| --- | --- |
| [http](https://nodejs.org/api/http.html) | http module includes classes, methods and events to create Node.js http server. |
| [url](https://nodejs.org/api/url.html) | url module includes methods for URL resolution and parsing. |
| [querystring](https://nodejs.org/api/querystring.html) | querystring module includes methods to deal with query string. |
| [path](https://nodejs.org/api/path.html) | path module includes methods to deal with file paths. |
| [fs](https://nodejs.org/api/fs.html) | fs module includes classes, methods, and events to work with file I/O. |
| [util](https://nodejs.org/api/util.html) | util module includes utility functions useful for programmers. |

### Loading Core Modules

In order to use Node.js core or NPM modules, you first need to import it using require() function as shown below.

var module = require('module\_name');

As per above syntax, specify the module name in the require() function. The require() function will return an object, function, property or any other JavaScript type, depending on what the specified module returns.

The following example demonstrates how to use Node.js http module to create a web server.

Example: Load and Use Core http Module

 Copy

var http = require('http');

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

//write code here

});

server.listen(5000);

In the above example, require() function returns an object because http module returns its functionality as an object, you can then use its properties and methods using dot notation e.g. http.createServer().

In this way, you can load and use Node.js core modules in your application. We will be using core modules throughout these tutorials.

Learn about local modules in the next section.

Node.js Local Module

Local modules are modules created locally in your Node.js application. These modules include different functionalities of your application in separate files and folders. You can also package it and distribute it via NPM, so that Node.js community can use it. For example, if you need to connect to MongoDB and fetch data then you can create a module for it, which can be reused in your application.

Writing Simple Module

Let's write simple logging module which logs the information, warning or error to the console.

In Node.js, module should be placed in a separate JavaScript file. So, create a Log.js file and write the following code in it.

Log.js

 Copy

var log = {

info: function (info) {

console.log('Info: ' + info);

},

warning:function (warning) {

console.log('Warning: ' + warning);

},

error:function (error) {

console.log('Error: ' + error);

}

};

module.exports = log

In the above example of logging module, we have created an object with three functions - info(), warning() and error(). At the end, we have assigned this object to **module.exports**. The module.exports in the above example exposes a log object as a module.

The *module.exports* is a special object which is included in every JS file in the Node.js application by default. Use **module.exports** or **exports** to expose a function, object or variable as a module in Node.js.

Now, let's see how to use the above logging module in our application.

Loading Local Module

To use local modules in your application, you need to load it using require() function in the same way as core module. However, you need to specify the path of JavaScript file of the module.

The following example demonstrates how to use the above logging module contained in Log.js.

app.js

 Copy

var myLogModule = require('./Log.js');

myLogModule.info('Node.js started');

In the above example, app.js is using log module. First, it loads the logging module using require() function and specified path where logging module is stored. Logging module is contained in Log.js file in the root folder. So, we have specified the path './Log.js' in the require() function. The '.' denotes a root folder.

The require() function returns a log object because logging module exposes an object in Log.js using module.exports. So now you can use logging module as an object and call any of its function using dot notation e.g myLogModule.info() or myLogModule.warning() or myLogModule.error()

Run the above example using command prompt (in Windows) as shown below.

C:\> node app.js  
Info: Node.js started

Thus, you can create a local module using module.exports and use it in your application.

Let's see how to expose different types as a node module using module.exports in the next section.

Export Module in Node.js

Here, you will learn how to expose different types as a module using module.exports.

The module.exports is a special object which is included in every JavaScript file in the Node.js application by default. The module is a variable that represents the current module, and exports is an object that will be exposed as a module. So, whatever you assign to module.exports will be exposed as a module.

Let's see how to expose different types as a module using module.exports.

Export Literals

As mentioned above, exports is an object. So it exposes whatever you assigned to it as a module. For example, if you assign a string literal then it will expose that string literal as a module.

The following example exposes simple string message as a module in Message.js.

Message.js

 Copy

module.exports = 'Hello world';

Now, import this message module and use it as shown below.

app.js

 Copy

var msg = require('./Messages.js');

console.log(msg);

Run the above example and see the result, as shown below.

C:\> node app.js  
Hello World

 Note:

You must specify ./ as a path of root folder to import a local module. However, you do not need to specify the path to import Node.js core modules or NPM modules in the require() function.

Export Object

The exports is an object. So, you can attach properties or methods to it. The following example exposes an object with a string property in Message.js file.

Message.js

 Copy

exports.SimpleMessage = 'Hello world';

//or

module.exports.SimpleMessage = 'Hello world';

In the above example, we have attached a property SimpleMessage to the exports object. Now, import and use this module, as shown below.

app.js

 Copy

var msg = require('./Messages.js');

console.log(msg.SimpleMessage);

In the above example, the require() function will return an object { SimpleMessage : 'Hello World'} and assign it to the msg variable. So, now you can use msg.SimpleMessage.

Run the above example by writing node app.js in the command prompt and see the output as shown below.

C:\> node app.js  
Hello World

In the same way as above, you can expose an object with function. The following example exposes an object with the log function as a module.

Log.js

 Copy

module.exports.log = function (msg) {

console.log(msg);

};

The above module will expose an object- { log : function(msg){ console.log(msg); } } . Use the above module as shown below.

app.js

 Copy

var msg = require('./Log.js');

msg.log('Hello World');

Run and see the output in command prompt as shown below.

C:\> node app.js  
Hello World

You can also attach an object to module.exports, as shown below.

data.js

 Copy

module.exports = {

firstName: 'James',

lastName: 'Bond'

}

app.js

 Copy

var person = require('./data.js');

console.log(person.firstName + ' ' + person.lastName);

Run the above example and see the result, as shown below.

C:\> node app.js  
James Bond

ADVERTISEMENT

Export Function

You can attach an anonymous function to exports object as shown below.

Log.js

 Copy

module.exports = function (msg) {

console.log(msg);

};

Now, you can use the above module, as shown below.

app.js

 Copy

var msg = require('./Log.js');

msg('Hello World');

The msg variable becomes a function expression in the above example. So, you can invoke the function using parenthesis (). Run the above example and see the output as shown below.

C:\> node app.js  
Hello World

Export Function as a Class

In JavaScript, a function can be treated like a class. The following example exposes a function that can be used like a class.

Person.js

 Copy

module.exports = function (firstName, lastName) {

this.firstName = firstName;

this.lastName = lastName;

this.fullName = function () {

return this.firstName + ' ' + this.lastName;

}

}

The above module can be used, as shown below.

app.js

 Copy

var person = require('./Person.js');

var person1 = new person('James', 'Bond');

console.log(person1.fullName());

As you can see, we have created a person object using the new keyword. Run the above example, as shown below.

C:\> node app.js  
James Bond

In this way, you can export and import a local module created in a separate file under root folder.

Node.js also allows you to create modules in sub folders. Let's see how to load module from sub folders.

Load Module from the Separate Folder

Use the full path of a module file where you have exported it using module.exports. For example, if the log module in the log.js is stored under the utility folder under the root folder of your application, then import it, as shown below.

app.js

 Copy

var log = require('./utility/log.js');

In the above example, . is for the root folder, and then specify the exact path of your module file. Node.js also allows us to specify the path to the folder without specifying the file name. For example, you can specify only the utility folder without specifying log.js, as shown below.

app.js

 Copy

var log = require('./utility');

In the above example, Node.js will search for a package definition file called package.json inside the utility folder. This is because Node assumes that this folder is a package and will try to look for a package definition. The package.json file should be in a module directory. The package.json under utility folder specifies the file name using the main key, as shown below.

./utility/package.json

 Copy

{

"name" : "log",

"main" : "./log.js"

}

Now, Node.js will find the log.js file using the main entry in package.json and import it.

 Note:

If the package.json file does not exist, then it will look for index.js file as a module file by default.

NPM - 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.

https://www.tutorialsteacher.com/Content/images/tips.pngIt 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.

C:\> npm -v  
2.11.3

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

C:\> npm install npm -g

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

C:\> 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.

C:\>npm install <package name>

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

C:\MyNodeProj> npm install express

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.

C:\MyNodeProj> npm install express --save

The 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.

Node.js Web Server

In this section, we will learn how to create a simple Node.js web server and handle HTTP requests.

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.

server.js

 Copy

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.

server.js

 Copy

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.

ADVERTISEMENT

Now, run the above web server as shown below.

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

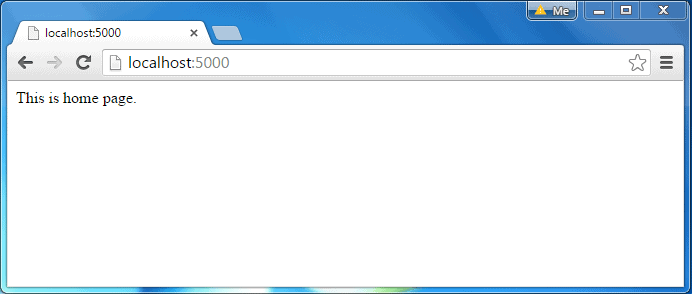
To test it, you can use the command-line program curl, which most Mac and Linux machines have pre-installed.

curl -i http://localhost:5000

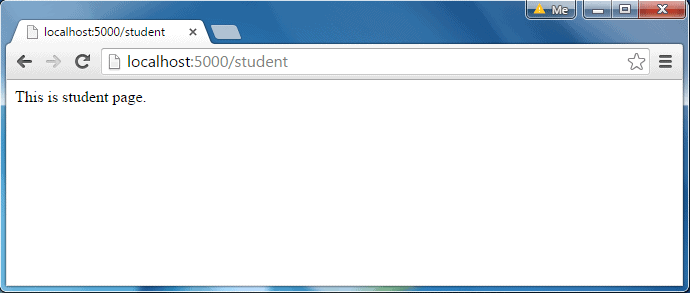
You should see the following response.

HTTP/1.1 200 OK  
Content-Type: text/plain  
Date: Tue, 8 Sep 2015 03:05:08 GMT  
Connection: keep-alive  
This is home page.

For Windows users, point your browser to *http://localhost:5000* and see the following result.

[](https://www.tutorialsteacher.com/Content/images/nodejs/webserver-response.png)Node.js Web Server Response

The same way, point your browser to *http://localhost:5000/student* and see the following result.

[](https://www.tutorialsteacher.com/Content/images/nodejs/webserver-response2.png)Node.js Web Server Response

It will display "Invalid Request" for all requests other than the above URLs.

Sending JSON Response

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

server.js

 Copy

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..')

So, this way you can create a simple web server that serves different responses.

# UNIT-4

# Node.js 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.

Signature:

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.

Example: Reading File

 Copy

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.

TextFile.txt

 Copy

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.

Example: Reading File Synchronously

 Copy

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.

Signature:

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.

Example: Creating & Writing File

 Copy

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.

Example: Append File Content

 Copy

var fs = require('fs');

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

if (err)

console.log(err);

else

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

});

ADVERTISEMENT

## Open File

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

Signature:

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.

Example:File open and read

 Copy

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.

Signature:

fs.unlink(path, callback);

The following example deletes an existing file.

Example:File Open and Read

 Copy

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. |

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.

The following example demonstrates EventEmitter class for raising and handling a custom event.

Example: Raise and Handle Node.js events

 Copy

// 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.

Example: EventEmitter

 Copy

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 following table lists all the important methods of EventEmitter class.

| EventEmitter Methods | Description |
| --- | --- |
| [emitter.addListener(event, listener)](https://nodejs.org/api/events.html#events_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)](https://nodejs.org/api/events.html#events_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)](https://nodejs.org/api/events.html#events_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)](https://nodejs.org/api/events.html#events_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])](https://nodejs.org/api/events.html#events_emitter_removealllisteners_event) | Removes all listeners, or those of the specified event. |
| [emitter.setMaxListeners(n)](https://nodejs.org/api/events.html#events_emitter_setmaxlisteners_n) | By default EventEmitters will print a warning if more than 10 listeners are added  For a particular event. |
| [emitter.getMaxListeners()](https://nodejs.org/api/events.html#events_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)](https://nodejs.org/api/events.html#events_emitter_listeners_event) | Returns a copy of the array of listeners for the specified event. |
| [emitter.emit(event[, arg1][, arg2][, ...])](https://nodejs.org/api/events.html#events_emitter_emit_event_arg1_arg2) | Raise the specified events with the supplied arguments. |
| [emitter.listenerCount(type)](https://nodejs.org/api/events.html#events_emitter_listenercount_type) | Returns the number of listeners listening to the type of event. |

C:\MyNodeP

# Express.js 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 Web Server

 Copy

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.

Example: Configure Routes in Express.js

 Copy

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');

});

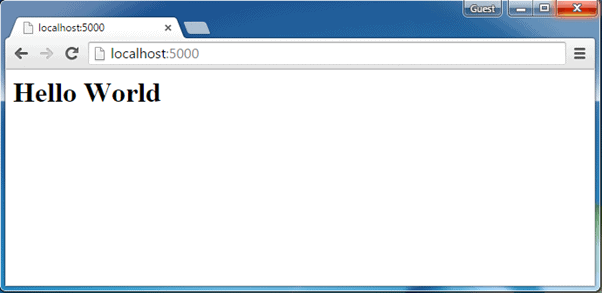
var server = app.listen(5000, function () {

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](https://expressjs.com/4x/api.html#req) and [response](https://expressjs.com/4x/api.html#res) 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.

[](https://www.tutorialsteacher.com/Content/images/nodejs/expressjs-webapp1.png)Express.js Web Application

ADVERTISEMENT

## 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.

Example: Configure Routes in Express.js

 Copy

<!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](https://github.com/expressjs/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.

app.js: Handle POST Route in Express.js

 Copy

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!');

});

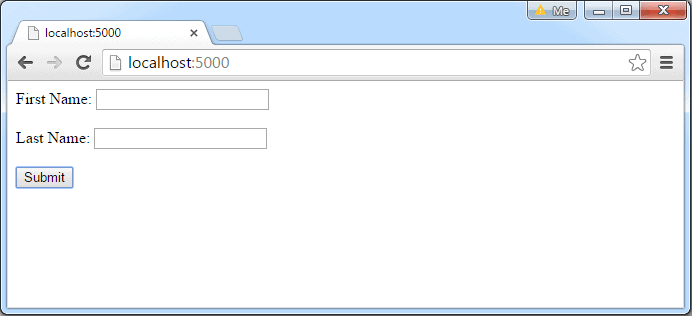
var server = app.listen(5000, function () {

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

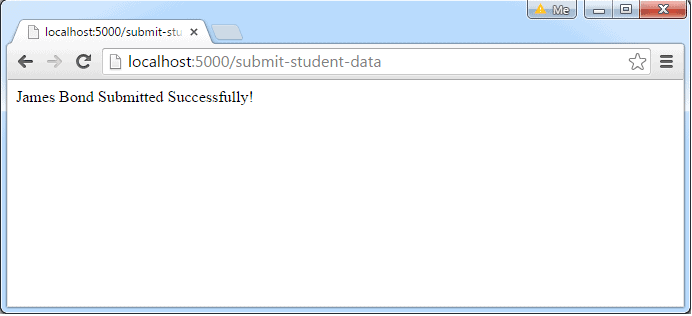
});

In 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.

[](https://www.tutorialsteacher.com/Content/images/nodejs/expressjs-post-request1.png)HTML Form to submit POST request

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. The following result is displayed.

[](https://www.tutorialsteacher.com/Content/images/nodejs/expressjs-post-data.png)Response from POST request

This is how you can handle HTTP requests using Express.js.

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.

Serve Static Resources using Express.js

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.

server.js

 Copy

var express = require('express');

var app = express();

//setting middleware

app.use(express.static(\_\_dirname + 'public')); //Serves resources from public folder

var server = app.listen(5000);

https://www.tutorialsteacher.com/Content/images/tips.pngSpecify absolute path in express.static() by prepending \_\_dirname. This will not break your application even if you run the express app from another directory.

In the above example, app.use() method mounts the middleware express.static for every request. The [express.static](https://expressjs.com/guide/using-middleware.html" \l "middleware.built-in" \t "_blank) middleware is responsible for serving the static assets of an Express.js application. The express.static() method specifies the folder from which to serve all static resources.

Now, run the above code using node server.js command and point your browser to *http://localhost:5000/myImage.jpg* and it will display myImage.jpg from the public folder (public folder should have myImage.jpg).

If you have different folders for different types of resources then you can set express.static middleware as shown below.

Example: Serve resources from different folders

 Copy

var express = require('express');

var app = express();

app.use(express.static('public'));

//Serves all the request which includes /images in the url from Images folder

app.use('/images', express.static(\_\_dirname + '/Images'));

var server = app.listen(5000);

In the above example, app.use() method mounts the express.static middleware for every request that starts with "/images". It will serve images from images folder for every HTTP requests that starts with "/images". For example, HTTP request http://localhost:5000/images/myImage.png will get myImage.png as a response. All other resources will be served from public folder.

Now, run the above code using node server.js and point your browser to *http://localhost:5000/images/myImage.jpg* and it will display myImage.jpg from the **images** folder, whereas *http://localhost:5000/myJSFile.js* request will be served from public folder. (images folder must include myImage.png and public folder must include myJSFile.js)

You can also create a virtual path in case you don't want to show actual folder name in the url.

Example: Setting virtual path

 Copy

app.use('/resources',express.static(\_\_dirname + '/images'));

So now, you can use *http://localhost:5000/resources/myImage.jpg* to serve all the images instead of *http://localhost:5000/images/myImage.jpg*.

In this way, you can use Express.js to server static resources such as images, CSS, JavaScript or other files.

ADVERTISEMENT

Serve Static Resources using Node-static Module

In your node application, you can use node-static module to serve static resources. The node-static module is an HTTP static-file server module with built-in caching.

First of all, install node-static module using NPM as below.

npm install node-static

After installing node-static module, you can create static file server in Node.js which serves static files only.

The following example demonstrates serving static resources using node-static module.

Example: Serving static resources using node-static

 Copy

var http = require('http');

var nStatic = require('node-static');

var fileServer = new nStatic.Server('./public');

http.createServer(function (req, res) {

fileServer.serve(req, res);

}).listen(5000);

In the above example, node-static will serve static files from public folder by default. So, an URL request will automatically map to the file in the public folder and will send it as a response.

Now, run the above example using node server.js command and point your browser to *http://localhost:5000/myImage.jpg* (assuming that public folder includes myImage.jpg file) and it will display the image on your browser. You don't need to give "/public/myImage.jpg" because it will automatically serve all the static files from the public folder.r