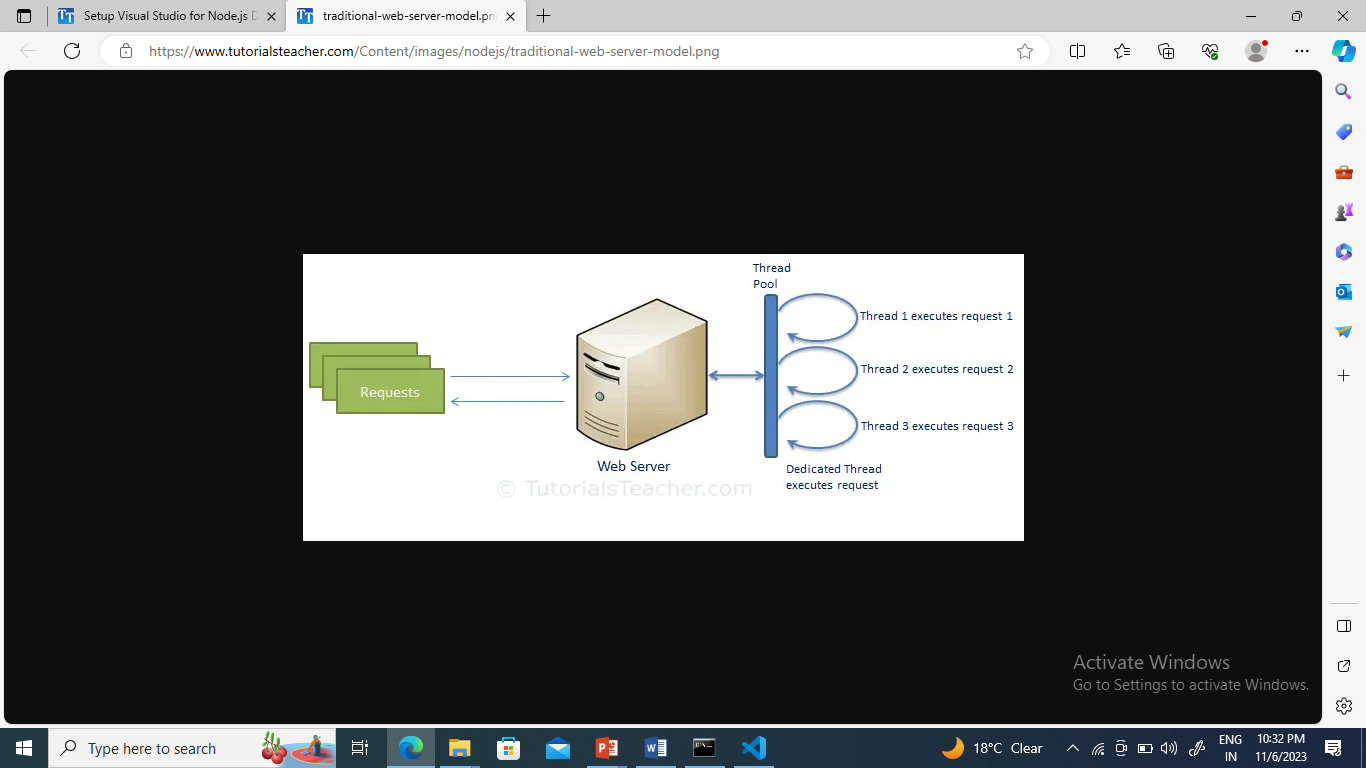
## **Node JS Process Model**

## **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.

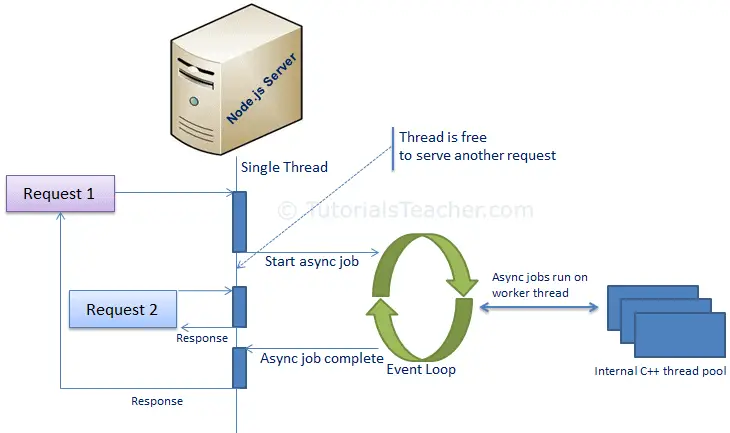


## **Node.js Process Model**

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 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.

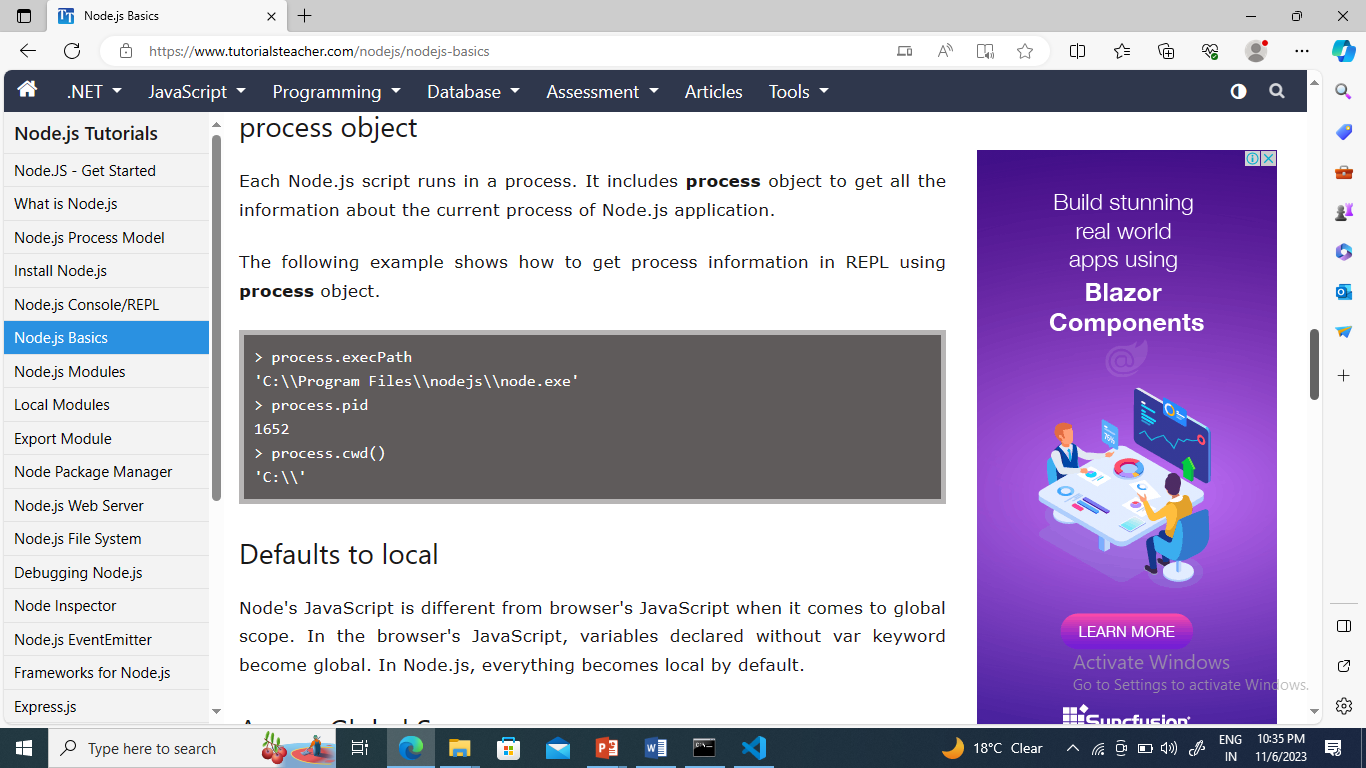
**Node Basics**

## **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.



## **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.**

|  |
| --- |
| 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.

# 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:-** 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**

|  |
| --- |
| 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().

# 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**

|  |
| --- |
| 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 export 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 mo

dule contained in Log.js.

**app.js**

|  |
| --- |
| 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) by giving the command:-

**Cmd> node app.js**

# ->Export Module in Node.js

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.

## **->Export Literals**

**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

|  |
| --- |
| module.exports = 'Hello world'; |

app.js

|  |
| --- |
| var msg = require('./Message.js');  console.log(msg); |

Run the above example in terminal

Cmd> node app.js

## **->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**

|  |
| --- |
| 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**

|  |
| --- |
| 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

Cmd> node app.js

## **->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

|  |
| --- |
| module.exports = function (firstName, lastName) {  this.firstName = firstName;  this.lastName = lastName;  this.fullName = function () {  return this.firstName + ' ' + this.lastName;  }  } |

**app.js**

|  |
| --- |
| 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

Cmd> node app.js

Node.js also allows you to create modules in 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**

|  |
| --- |
| 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**

|  |
| --- |
| 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**

|  |
| --- |
| {  "name" : "log",  "main" : "./log.js"  } |

Now, Node.js will find the log.js file using the main entry in package.json and import it.If the package.json file does not exist, then it will look for index.js file as a module file by default.

**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.common I/O operation using fs module are:-

## **->Reading a File:-**

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

**Syntax:-**

|  |
| --- |
| 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 file asynchronously.

**result.js**

|  |
| --- |
| 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.

**TextFile.txt**

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

Use the 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 a File**

Use the 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.

**Syntax:-**

|  |
| --- |
| 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.');  }); |

**->use the 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 the fs.open() method.

**Syntax:-**

|  |
| --- |
| 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. |

**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 the **fs.unlink()** method to delete an existing file.

**Syntax:-**

|  |
| --- |
| fs.unlink(path, callback); |

**Example:- deletes an existing file.**

|  |
| --- |
| var fs = require('fs');  fs.unlink('test.txt', function () {  console.log('File Deleted Successfully.');  }); |

## **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.unlink(path, callback); | Delete a file. |
| 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. |