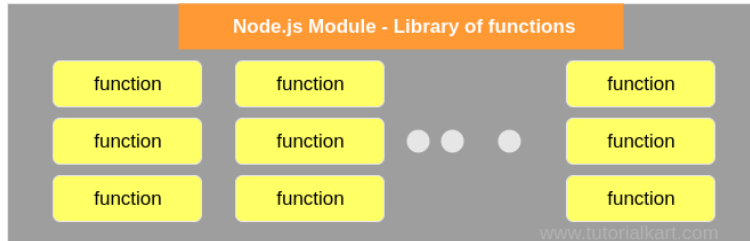


## Node.js Modules

In the Node.js module system, each file is treated as a separate module. A Node.js Module is a library of functions that could be used in a Node.js file.



There are three types of modules in Node.js based on their location to access. They are:

1. **Built-in Modules/Core Modules:** These are the modules that come along with Node.js installation.
2. **User defined Modules:** These are the modules written by user or a third party.
3. **Third party Modules:** There are many modules available online which could be used in Node.js. Node Package Manager (NPM) helps to install those modules, extend them if necessary and publish them to repositories like Github for access to distributed machines.

- Install a Node.js module using NPM
- Extend a Node.js module
- Publish a Node.js module to Github using NPM

---

### 1. Built-in Modules/Core Modules:

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	http module includes classes, methods and events to create Node.js http server.
url	url module includes methods for URL resolution and parsing.
querystring	querystring module includes methods to deal with query string.
path	path module includes methods to deal with file paths.
fs	fs module includes classes, methods, and events to work with file I/O.
util	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.

```
var http = require('http');

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

  console.log('http demo')
  res.end('hello world')

});

server.listen(5000);
```

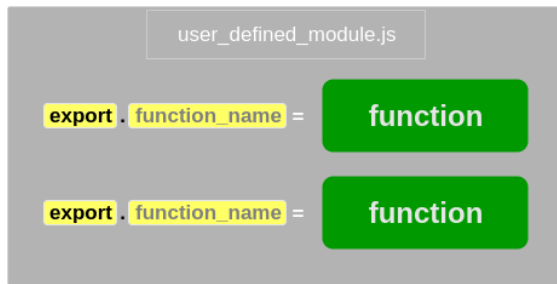
In the above example, require() function returns an object because http module returns its functionality as an object, you can then use its properties and methods using dot notation e.g. http.createServer().

---

## 2. User defined Modules

- How to Create a Node.js Module

Most of the necessary functions are included in the Built-in Modules. Sometimes it is required that, when you are implementing a Node.js application for a use case, you might want to keep your business logic separately. In such cases you create a Node.js module with all the required functions in it.



## Create a Node.js Module

A Node.js Module is a .js file with one or more functions.

Following is the syntax to define a function in Node.js module :

```
exports.<function_name> = function (argument_1, argument_2, .. argument_N) { /** function  
body */ };
```

**exports** – It is a keyword which tells Node.js that the function is available outside the module.

Example:

<u>circle.js</u>	<u>foo.js</u>
<pre>const {PI}=Math exports.f1= function (r) {   return PI * r ** 2; }  exports.datetime= function() {   return Date(); }  var x1=function() {   return 4 * 5 } exports.s=x1;  exports.multiplies6 = (a,b) =&gt; {   console.log('from arrow test');</pre>	<pre>var app = require('./circle.js');  console.log(`circle area of 4 is \${app.f1(4)}`);  console.log("date is " +app.datetime());  console.log('Multiplication '+app.s())  console.log('arrow function ' +app.multiplies6(5,3));</pre>

<pre>    return a * b   };   console.log('from circle '+x1());</pre>	
--	--

### Exports an object:

Ex:

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);
  }
};

var str="hello world";

exports.str=str;

exports.sr = log
```

app.js

```
var myLogModule = require('./log');

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

console.log(myLogModule.str);
```

Output:

H:\node\myexamples>node app.js

Info: Node.js started

hello world

Here, *var* log is an object. Info, warning and error are function. 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.

---

### 3. Third party Modules:

- The 3rd party modules can be downloaded using NPM (Node Package Manager).
- 3rd party modules can be install inside the project folder or globally.

Load and Use Third Party Module with Example:

3rd party modules can be downloaded using NPM in following way.

```
1. //Syntax
2.
3. npm install -g module_name // Install Globally
4.
5. npm install --save module_name //Install and save in package.json
6.
7.
8.
9. //Install express module
10.
11. npm install --save express
12.
13. npm install --save mongoose
14.
15.
16.
17. //Install multiple module at once
18.
19. npm install --save express mongoos
```

---

### HTTP Module

Node.js has a built-in module called HTTP, which allows Node.js to transfer data over the Hyper Text Transfer Protocol (HTTP).

### **Node.js as a Web Server**

The HTTP module can create an HTTP server that listens to server ports and gives a response back to the client.

Example:

```
var http = require('http');

//create a server object:
http.createServer(function (req, res) {
  res.write('Hello World!'); //write a response to the client
  res.end(); //end the response
}).listen(8080); //the server object listens on port 8080
```

The function passed into the `http.createServer()` method, will be executed when someone tries to access the computer on port 8080. We are using the `server.listen` function to make our server application listen to client requests on port no 8080. You can specify any available port over here.

### **Add an HTTP Header**

If the response from the HTTP server is supposed to be displayed as HTML, you should include an HTTP header with the correct content type:

Example

```
var http = require('http');
http.createServer(function (req, res) {
  res.writeHead(200, {'Content-Type': 'text/html'});
  res.write('Hello World!');
  res.end();
}).listen(8080);
```

The first argument of the `res.writeHead()` method is the status code, 200 means that all is OK, the second argument is an object containing the response headers.

---

### **Status-Code**

The Status-Code element in a server response, is a 3-digit integer where the first digit of the Status-Code defines the class of response and the last two digits do not have any categorization role. There are 5 values for the first digit:

S.N.	Code and Description
------	----------------------

1	1xx: Informational  It means the request has been received and the process is continuing.
2	2xx: Success  It means the action was successfully received, understood, and accepted.
3	3xx: Redirection  It means further action must be taken in order to complete the request.
4	4xx: Client Error  It means the request contains incorrect syntax or cannot be fulfilled.
5	5xx: Server Error  It means the server failed to fulfill an apparently valid request.

HTTP status codes are extensible and HTTP applications are not required to understand the meaning of all the registered status codes. Given below is a list of all the status codes.

### Read the Query String

The function passed into the `http.createServer()` has a `req` argument that represents the request from the client, as an object (`http.IncomingMessage` object).

This object has a property called `"url"` which holds the part of the url that comes after the domain name:

`demo_http_url.js`

```
var http = require('http');
http.createServer(function (req, res) {
  res.writeHead(200, {'Content-Type': 'text/html'});
  res.write(req.url);
  res.end();
}).listen(8080);
```

Save the code above in a file called `"demo_http_url.js"` and initiate the file:

Initiate demo\_http\_url.js:

```
C:\Users\BNJ>node demo_http_url.js
```

If you have followed the same steps on your computer, you should see two different results when opening these two addresses:

<http://localhost:8080/UVPCE>

Will produce this result:

/UVPCE

<http://localhost:8080/Ganpat>university

Will produce this result:

/ Ganpatuniversity

## Split the Query String

There are built-in modules to easily split the query string into readable parts, such as the URL module.

Example

Split the query string into readable parts:

```
var http = require('http');
```

```
var url = require('url');
```

```
http.createServer(function (req, res) {  
  res.writeHead(200, {'Content-Type': 'text/html'});  
  var q = url.parse(req.url, true).query;  
  var txt = q.year + " " + q.month;  
  res.end(txt);  
}).listen(8080);
```

Save the code above in a file called "demo\_querystring.js" and initiate the file:

```
C:\Users\Your Name>node demo_querystring.js
```

The address:

<http://localhost:8080/?year=2020&month=July>

Will produce this result:

2020 July



**Note:**

The **url.parse()** method takes a URL string, parses it, and it will return a URL object with each part of the address as properties.

**Syntax:**

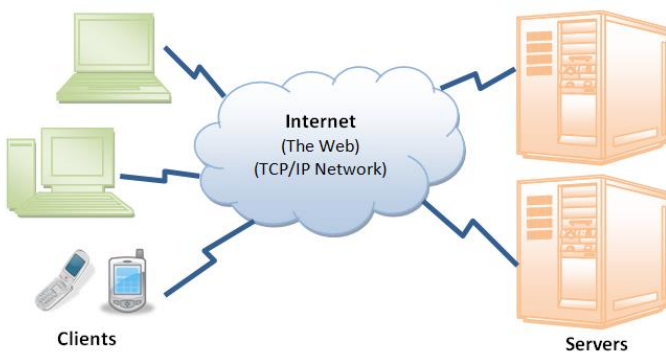
```
url.parse( urlString, parseQueryString, slashesDenoteHost)
```

**Parameters:** This method accepts three parameters as mentioned above and described below:

- **urlString:** It holds the URL string which needs to be parsed.
- **parseQueryString:** It is a boolean value. If it is set to true then the query property will be set to an object returned by the querystring module's parse() method. If it is set to false then the query property on the returned URL object will be an unparsed, undecoded string. Its default value is false.
- **slashesDenoteHost:** It is a boolean value. If it is set to true then the first token after the literal string // and preceding the next / will be interpreted as the host. For example: // https://www.ganpatuniversity.ac.in/exams/cbcs-regulations contains the result {host: 'www.ganpatuniversity.ac.in', pathname: '/exams/cbcs-regulations'} rather than {pathname: '//www.ganpatuniversity.ac.in/exams/cbcs-regulations'}. Its default value is false.

**Return Value:** The **url.parse()** method returns an object with each part of the address

## HTTP (HyperText Transfer Protocol)



As mentioned, whenever you enter a URL in the address box of the browser, the browser translates the URL into a request message according to the specified protocol; and sends the request message to the server.

For example, the browser translated the URL `http://www.nowhere123.com/doc/index.html` into the following request message:

```
GET /docs/index.html HTTP/1.1
Host: www.nowhere123.com
Accept: image/gif, image/jpeg, */*
Accept-Language: en-us
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1)
(blank line)
```

When this request message reaches the server, the server can take either one of these actions:

1. The server interprets the request received, maps the request into a *file* under the server's document directory, and returns the file requested to the client.
2. The server interprets the request received, maps the request into a *program* kept in the server, executes the program, and returns the output of the program to the client.
3. The request cannot be satisfied, the server returns an error message.

An example of the HTTP response message is as shown:

```
HTTP/1.1 200 OK
Date: Sun, 18 Oct 2009 08:56:53 GMT
Server: Apache/2.2.14 (Win32)
Last-Modified: Sat, 20 Nov 2004 07:16:26 GMT
ETag: "10000000565a5-2c-3e94b66c2e680"
Accept-Ranges: bytes
Content-Length: 44
Connection: close
Content-Type: text/html
X-Pad: avoid browser bug
```

```
<html><body><h1>It works!</h1></body></html>
```

The browser receives the response message, interprets the message and displays the contents of the message on the browser's window according to the media type of the response (as in the Content-Type response header). Common media type include "text/plain", "text/html", "image/gif", "image/jpeg", "audio/mpeg", "video/mpeg", "application/msword", and "application/pdf".

In its idling state, an HTTP server does nothing but listening to the IP address(es) and port(s) specified in the configuration for incoming request. When a request arrives, the server analyzes the message header, applies rules specified in the configuration, and takes the appropriate action. The webmaster's main control over the action of web server is via the configuration, which will be dealt with in greater details in the later sections.

---

**File System:** To handle file operations like creating, reading, deleting, etc., Node.js provides an inbuilt module called FS (File System). Node.js gives the functionality of file I/O by providing wrappers around the standard POSIX(IEEE STANDARD) functions. All file system operations can have synchronous and asynchronous forms depending upon user requirements.

The `process` object in Node.js is a global object that can be accessed inside any module without requiring it. There are very few global objects or properties provided in Node.js and `process` is one of them. It is an essential component in the Node.js ecosystem as it provides various information sets about the runtime of a program.

To explore we will use one of its properties which is called `process.versions`. This property tells us the information about Node.js version we have installed. It has to be used with `-p` flag.

```
$ node -p "process.versions"
```

```
# output
```

```
{ http_parser: '2.8.0',  
  node: '8.11.2',  
  v8: '6.2.414.54',  
  uv: '1.19.1',  
  zlib: '1.2.11',  
  ares: '1.10.1-DEV',  
  modules: '57',  
  nghttp2: '1.29.0',  
  napi: '3',  
  openssl: '1.0.2o',  
  icu: '60.1',  
  unicode: '10.0',  
  cldr: '32.0',  
  tz: '2017c' }
```

Another property you can check is `process.release` that is the same as the command `$ node --version` which we used when we installed Node.js. But the output this time is going to be more detailed.

```
node -p "process.release"
```

```
# output
```

```
{ name: 'node',  
  lts: 'Carbon',
```

```
    sourceUrl: 'https://nodejs.org/download/release/v8.11.2/node-v8.11.2.tar.gz',  
    headersUrl: 'https://nodejs.org/download/release/v8.11.2/node-v8.11.2-headers.tar.gz' }  
}
```

These are some of the different commands that we can use in a command line to access information that otherwise no module can provide.

This `process` object is an instance of the `EventEmitter` class. It does it contain its own pre-defined events such as `exit` which can be used to know when a program in Node.js has completed its execution.

Run the below program and you can observe that the result comes up with status code `0`. In Node.js this status code means that a program has run successfully.

```
process.on('exit', code => {  
    setTimeout(() => {  
        console.log('Will not get displayed');  
    }, 0);  
  
    console.log('Exited with status code:', code);  
});  
console.log('Execution Completed');
```

Output of the above program:

```
Execution Completed
```

```
Exited with status code: 0
```

`Process` also provides various properties to interact with. Some of them can be used in a Node application to provide a gateway to communicate between the Node application and any command line interface. This is very useful if you are building a command line application or utility using Node.js

- `process.stdin`: a readable stream
- `process.stdout`: a writable stream
- `process.stderr`: a writable stream to recognize errors

Using `argv` you can always access arguments that are passed in a command line. `argv` is an array which has Node itself as the first element and the absolute path of the file as the second element. From the third element onwards it can have as many arguments as you want.

Try the below program to get more insight into how you can use these various properties and functions.

```
process.stdout.write('Hello World!' + '\n');

process.argv.forEach(function(val, index, array) {
    console.log(index + ': ' + val);
});
```

If you run the above code with the following command you will get the output and the first two elements are of `argv` are printed.

```
$ node test.js
```

```
# output
Hello World!
0: /usr/local/bin/node
1: /Users/amanhimself/Desktop/articles/nodejs-text-tuts/test.js
```

## Buffer

`Buffer` objects are used to represent a fixed-length sequence of bytes. Many Node.js APIs support `Buffers`.

The `Buffer` class is a subclass of JavaScript's `Uint8Array` class and extends it with methods that cover additional use cases. Node.js APIs accept plain `Uint8Arrays` wherever `Buffers` are supported as well.

While the `Buffer` class is available within the global scope, it is still recommended to explicitly reference it via an `import` or `require` statement.

## What Are Buffers?

The `Buffer` class in Node.js is designed to handle raw binary data. Each buffer corresponds to some raw memory allocated outside V8. Buffers act somewhat like arrays of integers, but aren't resizable and have a whole bunch of methods specifically for binary data. The integers in a buffer each represent a byte and so are limited to values from 0 to 255 inclusive. When using `console.log()` to print the `Buffer` instance, you'll get a chain of values in hexadecimal values.

### Where You See Buffers:

In the wild, buffers are usually seen in the context of binary data coming from streams, such as `fs.createReadStream`.

### Usage:

#### Creating Buffers:

There are a few ways to create new buffers:

```
const buffer = Buffer.alloc(8);  
// This will print out 8 bytes of zero:  
// <Buffer 00 00 00 00 00 00 00 00>
```

This buffer is initialized and contains 8 bytes of zero.

```
const buffer = Buffer.from([8, 6, 7, 5, 3, 0, 9]);  
// This will print out 8 bytes of certain values:  
// <Buffer 08 06 07 05 03 00 09>
```

This initializes the buffer to the contents of this array. Keep in mind that the contents of the array are integers representing bytes.

```
const buffer = Buffer.from("I'm a string!", 'utf-8');  
// This will print out a chain of values in utf-8:  
// <Buffer 49 27 6d 20 61 20 73 74 72 69 6e 67 21>
```

This initializes the buffer to a binary encoding of the first string as specified by the second argument (in this case, `'utf-8'`). `'utf-8'` is

by far the most common encoding used with Node.js, but `Buffer` also supports others. See [Supported Encodings](#) for more details.

## Writing to Buffers

Given that there is already a buffer created:

```
> var buffer = Buffer.alloc(16)
```

we can start writing strings to it:

```
> buffer.write("Hello", "utf-8")
5
```

The first argument to `buffer.write` is the string to write to the buffer, and the second argument is the string encoding. It happens to default to utf-8 so this argument is extraneous.

`buffer.write` returned 5. This means that we wrote to five bytes of the buffer. The fact that the string "Hello" is also 5 characters long is coincidental, since each character *just happened* to be 8 bits apiece. This is useful if you want to complete the message:

```
> buffer.write(" world!", 5, "utf-8")
7
```

When `buffer.write` has 3 arguments, the second argument indicates an offset, or the index of the buffer to start writing at.

Reading from Buffers:

*toString:*

Probably the most common way to read buffers is to use the `toString` method, since many buffers contain text:

```
> buffer.toString('utf-8')
'Hello world!\u0000k\t'
```

Again, the first argument is the encoding. In this case, it can be seen that not the entire buffer was used! Luckily, because we know how

many bytes we've written to the buffer, we can simply add more arguments to "stringify" the slice that's actually interesting:

```
> buffer.toString("utf-8", 0, 12)
'Hello world!'
```

## setImmediate()

When you want to execute some piece of code asynchronously, but as soon as possible, one option is to use the `setImmediate()` function provided by Node.js:

```
setImmediate(() => {
```

```
  // run something
```

```
});
```

Any function passed as the `setImmediate()` argument is a callback that's executed in the next iteration of the event loop.

How is `setImmediate()` different from `setTimeout(() => {}, 0)` (passing a 0ms timeout), and from `process.nextTick()` and `Promise.then()`?

A function passed to `process.nextTick()` is going to be executed on the current iteration of the event loop, after the current operation ends. This means it will always execute before `setTimeout` and `setImmediate`.

A `setTimeout()` callback with a 0ms delay is very similar to `setImmediate()`. The execution order will depend on various factors, but they will be both run in the next iteration of the event loop.

A `process.nextTick` callback is added to `process.nextTick` queue.

A `Promise.then()` callback is added to promises microtask queue.

A `setTimeout`, `setImmediate` callback is added to macrotask queue.

Event loop executes tasks in `process.nextTick` queue first, and then executes promises microtask queue, and then executes macrotask queue.

Here is an example to show the order between `setImmediate()`, `process.nextTick()` and `Promise.then()`:

```
const baz = () => console.log('baz');
```



```
const foo = () => console.log('foo');
const zoo = () => console.log('zoo');
const start = () => {
  console.log('start');
  setImmediate(baz);
  new Promise((resolve, reject) => {
    resolve('bar');
  }).then((resolve) => {
    console.log(resolve);
    process.nextTick(zoo);
  });
  process.nextTick(foo);
};
start();
```

Output

```
Foo
Bar
Zoo
Bas
```

## setTimeout()

When writing JavaScript code, you might want to delay the execution of a function.

This is the job of `setTimeout`. You specify a callback function to execute later, and a value expressing how later you want it to run, in milliseconds:

```
setTimeout(() => {
  // runs after 2 seconds
}, 2000);
```

```
setTimeout(() => {
  // runs after 50 milliseconds
}, 50);
```

This syntax defines a new function. You can call whatever other function you want in there, or you can pass an existing function name, and a set of parameters:

`setTimeout` returns the timer id. This is generally not used, but you can store this id, and clear it if you want to delete this scheduled function execution:

```
const id = setTimeout(() => {
  // should run after 2 seconds
```

```
}, 2000);
```

```
// I changed my mind  
clearTimeout(id);
```

## Zero delay

If you specify the timeout delay to 0, the callback function will be executed as soon as possible, but after the current function execution:

```
setTimeout(() => {  
  console.log('after ');  
}, 0);
```

```
console.log(' before ');
```

This code will print

```
before  
after
```

This is especially useful to avoid blocking the CPU on intensive tasks and let other functions be executed while performing a heavy calculation, by queuing functions in the scheduler.

## setInterval()

`setInterval` is a function similar to `setTimeout`, with a difference: instead of running the callback function once, it will run it forever, at the specific time interval you specify (in milliseconds):

```
setInterval(() => {  
  // runs every 2 seconds  
}, 2000);
```

The function above runs every 2 seconds unless you tell it to stop, using `clearInterval`, passing it the interval id that `setInterval` returned:

```
const id = setInterval(() => {  
  // runs every 2 seconds  
}, 2000);
```

```
clearInterval(id);
```

It's common to call `clearInterval` inside the `setInterval` callback function, to let it auto-determine if it should run again or stop. For example this code runs something unless `App.somethingIWait` has the value `arrived`:

```
const interval = setInterval(() => {  
  if (App.somethingIWait === 'arrived') {  
    clearInterval(interval);  
  }  
  // otherwise do things  
, 100);
```

## FileSystem

To use this File System module, use the `require()` method:

```
var fs = require('fs');
```

Common use for File System module:

- Read Files
- Write Files
- Append Files
- Close Files
- Delete Files
- Rename file

What is **Synchronous** and **Asynchronous** approach?

**Synchronous approach:** They are called blocking functions as it waits for each operation to complete, only after that, it executes the next operation, hence blocking the next command from execution i.e. a command will not be executed until & unless the query has finished executing to get all the result from previous commands.

**Asynchronous approach:** They are called non-blocking functions as it never waits for each operation to complete, rather it executes all operations in the first go itself. The result of each operation will be handled once the result is available i.e. each command will be executed soon after the execution of the previous command. While the previous command runs in the background and loads the result once it is finished processing the data.

Use cases:

If your operations **are not doing very heavy lifting like querying huge data from DB** then go ahead with **Synchronous way** otherwise asynchronous way.

In an Asynchronous way, you can show some progress indicator to the user while in the background you can continue with your heavyweight works. This is an ideal scenario for GUI based apps.

## Reading File

Use `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". This field is optional.
- callback: A function with two parameters err and data. This will get called when readFile operation completes.

Example:

```
var fs = require("fs");

// Asynchronous read
fs.readFile('input.txt', function (err, data) {
  if (err) {
    return console.error(err);
  }
  console.log("Asynchronous read: " + data.toString());
});
```

## 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.

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.');
```

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

Example: Append File Content

```
var fs = require('fs');

fs.appendFile('test.txt', 'Hello World!', function (err) {
    if (err)
        console.log(err);
    else
        console.log('Append operation complete.');
```

## Delete File

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

Syntax :

```
fs.unlink(path, callback);
```

The following example deletes an existing file.

Example: Delete file

```
var fs = require('fs');

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

    console.log('Deleted successfully..');

});
```

## Renaming a File

A file can be renamed using `fs.rename()`. Here, it takes old file name, new file name, and a callback function as arguments.

```
fs.rename('hello1.txt', 'hello2.txt', function(err) {  
  if (err)  
    console.log(err);  
  
  console.log('Rename complete!');  
});
```

## For career objective

### What Is LeetCode?

It's a website where people—mostly software engineers—practice their coding skills. There are 800+ questions (and growing), each with multiple solutions. Questions are ranked by level of difficulty: easy, medium, and hard.

Similar websites include **HackerRank**, **Topcoder**, **InterviewBit**, among others. There's also a popular book, "Cracking the Coding Interview," which some call the Bible for engineers. The Blind community uses a mix of these resources, but based on mentions, LeetCode seems to be the most popular. Our active users cite the following reasons for preferring LeetCode: more questions, better quality, plus a strong user base.

Util module:

The node.js "util" module provides some functions to print formatted strings as well as some 'utility' functions that are helpful for debugging purposes. Use `require('util')` to access these functions. Following functions are in the module 'util'.

Contents:

`util.format(format, [...])`

`util.error([...])`

`util.print([...])`

`util.log(string)`

util.isArray(object)

util.isRegExp(object)

util.isDate(object)

util.isError(object)

util.inherits(constructor,superConstructor)

## util.format(format, [...])

The util.format() is used to create formatted string from one or more arguments. The first argument is a string that contains zero or more placeholders. A placeholder is a character sequence in the format string and is replaced by a different value in the returned string.

List of placeholder :

- % - Returns a single percent sign.
- d - Treated as Number (both integer and float).
- s - Treated as string and display as a string.
- j - Represent JSON data.

Here are some examples :

```
var util = require('util');
var my_name = 'Sunita',
    my_class = 5,
    my_roll_no = 11,
    my_fav_subject= { subj1: 'English', subj2: 'Math.'};
var format1 = util.format('My name is %s ',my_name);
var format2 = util.format('I read in class %d,',my_class);
var format3 = util.format('My roll no. is %d,',my_roll_no);
var format4 = util.format('My favorite subjects are %j',my_fav_subject);
console.log(format1);
console.log(format2);
console.log(format3);
console.log(format4);
```

Output :

```
E:\nodejs>node test.js
My name is Sunita
I read in class 5,
My roll no. is 11,
My favorite subjects are {"subj1":"English","subj2":"Math."}
```

## util.log(string)

The function is used to write the string out to stdout, with timestamp.

Here is an examples :

```
var util = require('util');
util.log('Timestamped message.');
```

Output :

```
E:\nodejs>node test.js
24 Oct 14:23:16 - Timestamped message.
```

## util.inspect(object, [options])

The function returns a string representation of object, which is useful for debugging.

Optional options :

- showHidden - if true then the object's non-enumerable properties will be shown too. Defaults to false.
- depth - tells inspect how many times to recurse while formatting the object. This is useful for inspecting large complicated objects. Defaults to 2. To make it recurse indefinitely pass null.
- colors - if true, then the output will be styled with ANSI color codes. Defaults to false. Colors are customizable, see below.
- customInspect - if false, then custom inspect() functions defined on the objects being inspected won't be called. Defaults to true.

The following example lists the Node's built-in objects.

```
var util = require('util')
console.log(util.inspect(console));
```

Output :

```
E:\nodejs>node test.js
{ log: [Function],
  info: [Function],
  warn: [Function],
  error: [Function],
  dir: [Function],
  time: [Function],
  timeEnd: [Function],
  trace: [Function],
  assert: [Function],
```



```
Console: [Function: Console] }
```

Here is an example of inspecting all properties of the util object :

```
var util = require('util');
console.log(util.inspect(util, { showHidden: true, depth: null }));
```

## util.isArray(object)

The function is used to check whether an 'object' is an array or not. Returns true if the given 'object' is an Array, false otherwise.

Here is an examples :

```
var util = require('util');
console.log(util.isArray([]));
console.log(util.isArray(new Array()));
console.log(util.isArray({}));
```

Output :

```
E:\nodejs>node test.js
true
true
false
```

## util.isRegExp(object)

The function is used to check whether an 'object' is RegExp or not. Returns true if the given 'object' is an RegExp, false otherwise.

Here is an examples :

```
var util = require('util');
console.log(util.isRegExp(/some regexp/));
onsole.log(util.isRegExp(new RegExp('New regexp')));
console.log(util.isRegExp({}));
```

Output :

```
E:\nodejs>node test.js
true
true
false
```

## util.isDate(object)

The function is used to check whether an 'object' is Date or not. Returns true if the given 'object' is an Date, false otherwise.

Here is an examples :

```
var util = require('util');
console.log(util.isDate(new Date()));
console.log(util.isDate(Date()));
console.log(util.isDate({}))
```

Output:

```
E:\nodejs>node test.js
true
false
false
```

### **util.isError(object)**

The function is used to check whether an 'object' is Error or not. Returns true if the given 'object' is an Error, false otherwise.

Here is an examples :

```
var util = require('util');
console.log(util.isError(new Error()));
console.log(util.isError(new TypeError()));
console.log(util.isError({ name: 'Error', message: 'an error occurred' }));
```

Output :

```
E:\nodejs>node test.js
true
true
false
```

### **The Node path module:**

These are the path methods:

path.basename()

path.dirname()

path.extname()

path.isAbsolute()

path.join()

`path.normalize()`

`path.parse()`

`path.relative()`

`path.resolve()`

**`path.basename()`**

Return the last portion of a path. A second parameter can filter out the file extension:

```
require('path').basename('/test/something') //something
require('path').basename('/test/something.txt') //something.txt
require('path').basename('/test/something.txt', '.txt') //something
```

**`path.dirname()`**

Return the directory part of a path:

```
require('path').dirname('/test/something') // /test
require('path').dirname('/test/something/file.txt') // /test/something
```

**`path.extname()`**

Return the extension part of a path

```
require('path').extname('/test/something') // ''
require('path').extname('/test/something/file.txt') // '.txt'
```

**`path.isAbsolute()`**

Returns true if it's an absolute path

```
require('path').isAbsolute('/test/something') // true
require('path').isAbsolute('./test/something') // false
```

**`path.join()`**

Joins two or more parts of a path:

```
const name = 'flavio'
require('path').join('/', 'users', name, 'notes.txt')
//'/users/flavio/notes.txt'
```

**`path.normalize()`**

Tries to calculate the actual path when it contains relative specifiers like `.` or `..`, or double slashes:

```
require('path').normalize('/users/flavio/../../test.txt') //users/test.txt
```

### **path.parse()**

Parses a path to an object with the segments that compose it:

- **root:** the root
- **dir:** the folder path starting from the root
- **base:** the file name + extension
- **name:** the file name
- **ext:** the file extension

Example:

```
require('path').parse('/users/test.txt')
```

results in

```
{
  root: '/',
  dir: '/users',
  base: 'test.txt',
  ext: '.txt',
  name: 'test'
}
```

### **path.relative()**

Accepts 2 paths as arguments. Returns the relative path from the first path to the second, based on the current working directory.

Example:

```
require('path').relative('/Users/flavio', '/Users/flavio/test.txt')
//'test.txt'
require('path').relative('/Users/flavio', '/Users/flavio/something/test.txt')
//'something/test.txt'
```

### **path.resolve()**

You can get the absolute path calculation of a relative path using `path.resolve()`:

```
path.resolve('flavio.txt') //'/Users/flavio/flavio.txt' if run from my home
folder
```

By specifying a second parameter, `resolve` will use the first as a base for the second:

```
path.resolve('tmp', 'flavio.txt')//'/Users/flavio/tmp/flavio.txt' if run from
my home folder
```

If the first parameter starts with a slash, that means it's an absolute path:

```
path.resolve('/etc', 'flavio.txt')//'/etc/flavio.txt'
```

### URL module:

*he URL module provides utilities for URL resolution and parsing. It can be accessed using:*

1. `var url = require('url');`

Url module is one of the core modules that comes with node.js, which is used to parse the URL and its other properties.

By using URL module, it provides us with so many properties to work with.

These all are listed below:

Property	Description
.href	Provides us the complete url string
.host	Gives us host name and port number
.hostname	Hostname in the url
.path	Gives us path name of the url
.pathname	Provides host name , port and pathname
.port	Gives us port number specified in url
.auth	Authorization part of url
.protocol	Protocol used for the request
.search	Returns query string attached with url

As you can see in the above screen, there are various properties used for URL module.

Below is the snippet to check the URL properties with URL : localhost:4200.

## MyApp.js

```
var http = require('http');

var url = require('url');

http.createServer(function (req, res) {

    var queryString = url.parse(req.url, true);

    console.log(queryString);

}).listen(4200);
```

Now run the above snippet using node MyApp.js, and you can see the console like this:

```
Url {
  protocol: null,
  slashes: null,
  auth: null,
  host: null,
  port: null,
  hostname: null,
  hash: null,
  search: '',
  query: {},
  pathname: '/',
  path: '/',
  href: '/' }
```

Because we do not have a path attached to URL, now I'm writing text within my URL like below:

1. <http://localhost:4200/bhavin>

And again I compiled the snippet and got output like this:

```
H:\BACKUP\5thceit>node url.js
Url {
  protocol: null,
  slashes: null,
  auth: null,
  host: null,
  port: null,
  hostname: null,
  hash: null,
  search: null,
  query: [Object: null prototype] {},
  pathname: '/bhavin',
  path: '/bhavin',
  href: '/bhavin'
}
```

So now, I got my complete pathname along with href as well as the path name and other properties with null values.

## href

Href property returns the complete URL along with all search terms and other information as well.

## href.js

```
var http = require('http');

var url = require('url');

http.createServer(function (req, res) {

  // Parsing url

  var queryString = url.parse(req.url,true);

  // Accessing href property of an URL

  console.log("Complete href is :-"+queryString.href);

}).listen(4200);
```

Execute the above snippet by writing node url.js and you can see the console like this:

```
H:\BACKUP\5thceit>node url.js
Complete href is :-/
Complete href is :-/bhavin
```

If we change our URL to [www.customway.com/abc.html](http://www.customway.com/abc.html):

```
var http = require('http');

var url = require('url');

http.createServer(function (req, res) {

    var queryString = url.parse(req.url,true);

    queryString.href = "https://www.google.com//uvpce.html";

    console.log("Complete href is :-"+queryString.href);

}).listen(4200);
```

And after that, you may get href like :

```
H:\BACKUP\5thceit>node url.js
Complete href is :-https://www.google.com//uvpce.html
Complete href is :-https://www.google.com//uvpce.html
```

So in this way we can get href from the requested URL

## **host and hostname**

Host property of a URL module provides the host associated with URL.

### **host.js**

```
var http = require('http');

const { URL } = require('url');
```



```
http.createServer(function (req, res) {  
  
    var queryString = url.parse(req.url,true);  
  
    // Prints the host  
  
    console.log("Host is :-"+queryString.host);  
  
    // Prints the host name  
  
    console.log("Host name is :-"+queryString.hostname);  
  
}).listen(4200);
```

After executing the above snippet you may get output like this:

```
PS C:\> node host.js  
Host is :-null  
Host name is :-null
```

### **What is the difference between host and hostname?**

There is one major difference between both of them, that is that host includes the port name along with hostname, where hostname property does not include the port number.

Now I'm going to create a new URL using the below snippet:

```
var http = require('http');  
  
const { URL } = require('url');  
  
http.createServer(function (req, res) {  
  
    const queryString1 = new URL('https://www.google.com:11/uvpce');  
  
    console.log("Host is :-"+queryString1.host);
```

```
console.log("Host name is :-"+queryString1.hostname);  
  
}).listen(4200);
```

So now, I have specified custom URL with port number 11, let's see the difference :

```
H:\BACKUP\5thceit>node url.js  
Host is :-www.google.com:11  
Host name is :-www.google.com
```

The host includes the port number, whereas hostname does not contain port number along with URL.

## Pathname and Searchparam

Path and pathname is the combination of pathname with URL and also contains a search term along with the URL

Let's say we have URL with multiple search parameters like :

- `www.demo.com/test1/test2/test3?qstring=value`

In above url:

- `path = /test1/test2/test3`
- `Search param = qstring=value`

In the same way we can get this via URL module property; find the snippet below :

### path.js

```
var http = require('http');  
  
const { URL } = require('url');  
  
http.createServer(function (req, res) {  
  
    const queryString2 = new  
    URL('https://www.google.com/test/test1/test2/test3?username=bhavin');  
  
    console.log("Path name is :-"+queryString2.pathname);
```

```
console.log("Search Parameter is :-"+queryString2.searchParams);  
}).listen(4200);
```

Now execute it by writing node path.js, and you will get output like this:

```
H:\BACKUP\5thceit>node url.js  
Path name is :-/test/test1/test2/test3  
Search Parameter is :-username=bhavin
```

This way you can access the complete path along with search parameters.

## Search

Search property is the same as searchParams that we have seen before

But the main difference is search property includes [?] along with all search parameters

### search.js

```
1. var http = require('http');  
2. const { URL } = require('url');  
3.  
4. http.createServer(function (req, res) {  
5.  
6.   const queryString = new URL('https://www.google.com/test/test1/test2/test3?username  
   =bhavin&password=123');  
7.   console.log("Search terms are :-"+queryString.search);  
8.  
9. }).listen(4200);
```

Now you will be able to get the whole search term along with [?] symbol attached :

```
H:\BACKUP\5thceit>node url.js  
Path name is :-/test/test1/test2/test3  
Search Parameter is :-?username=bhavin&pass=joshi  
Path name is :-/test/test1/test2/test3  
Search Parameter is :-?username=bhavin&pass=joshi
```

As you can see both of my search terms were included along with [?] symbol

## Port

Port property of a URL module returns the port associated with a URL.

### port.js

1. var http = require('http');
2. const { URL } = require('url');
- 3.
4. http.createServer(function (req, res) {
- 5.
6.     const queryString = new URL('https://www.customway.com:4200');
7.     console.log("Port is :-"+queryString.port);
- 8.
9. }).listen(4200);

I've used port number 4200 along with URL, and may get output like this:

```
PS C:\> node port.js
Port is :-4200
```

Now if I want to change port number, than we can do like this:

1. const queryString = new URL('https://www.customway.com:4200');
2.     queryString.port = '4500'; // changed port number to 4500
3.     console.log("Port is :-"+queryString.port);

And probably you will get output in the console:

- Port is :- 4500

## Protocol

Protocol property is used to get specific protocols used for any request.

### protocol.js

1. var http = require('http');
2. const { URL } = require('url');
- 3.
4. http.createServer(function (req, res) {
- 5.
6.     const queryString = new URL('https://www.customway.com');
7.     console.log("Protocol used :-"+ queryString.protocol);
- 8.
9. }).listen(4200);

Now, you will get the protocol name for what you have requested :

```
PS C:\> node protocol.js
Protocol used :-https:
```

## Hash

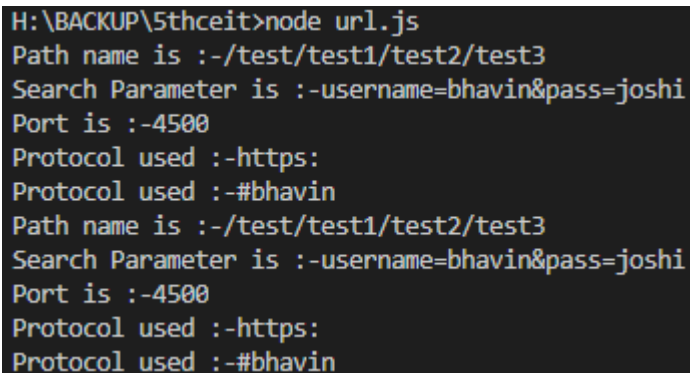
Hash property of a URL returns decorated with [#] symbol.

Sometimes, we have pages with multiple div, and we have provided ids along with #name, so in a node, we can also access hash fragment portion attached to URL.

### hash.js

```
1. var http = require('http');
2. const { URL } = require('url');
3.
4. http.createServer(function (req, res) {
5.
6.     const queryString = new URL('https://www.customway.com#bhavin');
7.     console.log("Hash Fragment Is :-" + queryString.hash);
8.
9. }).listen(4200);
```

After executing node hash.js you will get output like :



```
H:\BACKUP\5thceit>node url.js
Path name is :-/test/test1/test2/test3
Search Parameter is :-username=bhavin&pass=joshi
Port is :-4500
Protocol used :-https:
Protocol used :-#bhavin
Path name is :-/test/test1/test2/test3
Search Parameter is :-username=bhavin&pass=joshi
Port is :-4500
Protocol used :-https:
Protocol used :-#bhavin
```

Example:

```
var url = require('url');
var adr = 'http://localhost:8080/default.htm?year=2020&month=AUGUST';
var q = url.parse(adr, true);
```

```
console.log(q.host); //returns 'localhost:8080'
console.log(q.pathname); //returns '/default.htm'
console.log(q.search); //returns '?year=2020&month=AUGUST'
```

```
var qdata = q.query; //returns an object: { year: 2020, month: 'AUGUST' }  
console.log(qdata.month); //returns 'AUGUST'
```

x1.html

```
<html>  
<body>  
<h1>GANPAT UNIVERSITY</h1>  
<p>KHERVA MAHESANA</p>  
</body>  
</html>
```

X2.html

```
<html>  
<body>  
<h1>UVPCE</h1>  
<p> KHERVA MAHESANA </p>  
</body>  
</html>
```

Fileserver.js

```
var http = require('http');  
var url = require('url');  
var fs = require('fs');  
  
http.createServer(function (req, res) {  
  var q = url.parse(req.url, true);  
  var filename = "." + q.pathname;  
  fs.readFile(filename, function(err, data) {  
    if (err) {  
      res.writeHead(404, {'Content-Type': 'text/html'});  
      return res.end("404 Not Found");  
    }  
    res.writeHead(200, {'Content-Type': 'text/html'});  
    res.write(data);  
    return res.end();  
  });  
}).listen(8080);
```

<http://localhost:8080/x1.html>

<http://localhost:8080/x2.html>

## Express

### 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.

### Installing and using Express

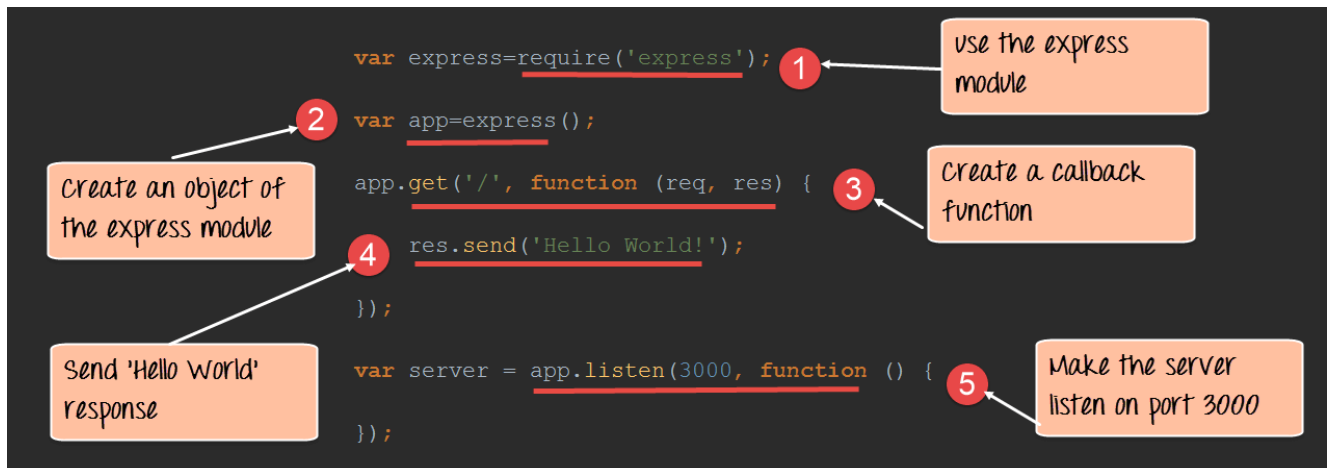
Express gets installed via the Node Package Manager. This can be done by executing the following line in the command line

**npm install express**

The above command requests the Node package manager to download the required express modules and install them accordingly.

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:

1. In our first line of code, we are using the require function to include the "express module."
2. Before we can start using the express module, we need to make an object of it.
3. 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.
4. 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.
5. 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.

### Output:



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.

### 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

1. A /Node route which will display the string "uvpcesite on Node" if this route is accessed
2. A /Angular route which will display the string "uvpcesite on Angular" if this route is accessed
3. A default route / which will display the string "Welcome to GANPAT UNIVERSITY."

```
const express1 = require('express')  
const app = express1()  
const port = 3000  
  
app.get('/', (req, res) => {  
  res.send('Welcome to GANPAT UNIVERSITY')  
})
```

```
app.route('/Node').get(function(req,res)
{
    res.send("uvpcesite on Node");
});
app.route('/Angular').get(function(req,res)
{
    res.send("uvpcesite on Angular");
});

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

    res.send('MY NAME IS BHAVIN');
})

app.listen(port, () => {
    console.log(`Example app listening at http://localhost:${port}`)
})
```

## NodeJS Frameworks

Let us now look at the popular NodeJs Frameworks:

### 1. Hapi.js



#### Features:

- Code reusability
- No external dependencies
- Security

- Integrated Architecture: comprehensive authorization and authentication API available in a node framework.

## 2. Express.js



Built by TJ Holowaychuk,

Express.js is a flexible and minimal Node.js application framework specifically designed for building single-page, multi-page, and hybrid applications that provide a robust set of features for web and mobile applications.

Express has no out-of-the-box object-relational mapping engine. Express isn't built around specific components, having "no opinion" regarding what technologies you plug into it. This freedom, coupled with lightning-fast setup and the pure JavaScript environment of Node, makes Express a strong candidate for agile development and rapid prototyping. Express is most popular with startups that want to build a product as quickly as possible and don't have very much legacy code.

The framework has the advantage of continuous updates and reforms of all the core features. It is a minimalist framework that is used to build several mobile applications and APIs.

## 3. Koa.js



#### 4. Sails.js



#### 5. Meteor.js



#### Features:

- Zero configuration build tools providing code splitting and dynamic imports.
- It is faster as it comes with real-time features.
- Nicely integrated frontend with backend
- Meteor methods that define server-side functionality on the server and then call the methods directly from the client-side and not have to interact with hidden API.
- Accounts and user authentication are excellent with meteor.
- Excellent platform for building as doesn't require code separate between its all a part of one code base that communicates smoothly.

#### 6. Derby.js



**Features:**

- Realtime Collaboration
- Server Rendering
- Components and data binding
- Modular

**7. Total.js**



**Features:**

- Rapid support and bug fixing
- Supports RESTful routing
- Supports video streaming
- Supports themes
- Supports workers
- Supports sitemap
- Supports WebSocket

- Supports models, modules, packages, and isomorphic code
- Supports Image processing via GM or IM
- Supports generators
- Supports localization with diff tool and CSV export
- Supports restrictions and redirections

## 8. Adonis.js



### Features:

- It has its own CLI (Command Line Interface)
- Familiar to Laravel so easy to learn
- Validators are used to check if the data flowing into the controllers has the right format, and emit messages when some errors occur.

## 9. Nest.js



**Features:**

- Extensible: Allows the use of any other libraries because of modular architecture, thus making it truly flexible.
- Versatile: It offers an adaptable ecosystem that is a fully-fledged backbone for all kinds of server-side applications.
- Progressive: Brings design patterns and sophisticated solutions to node.js world by taking advantage of the latest JavaScript features.

**10. LoopBack.js**



**Features:**

- Unbelievably extensible
- Graph QL Support

## Conclusion

Learning new frameworks is overwhelming and requires a lot of research before starting. Above mentioned frameworks are most popularly used and offer different features.

Here are some examples of route paths based on strings.

This route path will match requests to the root route, /.

```
app.get('/', function (req, res) {  
  res.send('root')  
})
```

This route path will match requests to /about.

```
app.get('/about', function (req, res) {  
  res.send('about')  
})
```

This route path will match requests to /random.text.

```
app.get('/random.text', function (req, res) {  
  res.send('random.text')  
})
```

Here are some examples of route paths based on string patterns.

This route path will match acd and abcd.

```
app.get('/ab?cd', function (req, res) {  
  res.send('ab?cd')  
})
```

This route path will match abcd, abbcd, abbbcd, and so on.

```
app.get('/ab+cd', function (req, res) {  
  res.send('ab+cd')  
})
```

This route path will match abcd, abxcd, abRANDOMcd, ab123cd, and so on.

```
app.get('/ab*cd', function (req, res) {
```



```
res.send('ab*cd')
})
```

This route path will match /abe and /abcde.

```
app.get('/ab(cd)?e', function (req, res) {
  res.send('ab(cd)?e')
})
```

Examples of route paths based on regular expressions:

This route path will match anything with an “a” in it.

```
app.get(/a/, function (req, res) {
  res.send('/a/')
})
```

This route path will match butterfly and dragonfly, but not butterflyman, dragonflyman, and so on.

```
app.get(/.*fly$/, function (req, res) {
  res.send('/.*fly$/')
})
```

## Middleware function myLogger

Here is a simple example of a middleware function called “myLogger”. This function just prints “LOGGED” when a request to the app passes through it. The middleware function is assigned to a variable named myLogger.

```
var myLogger = function (req, res, next) {
  console.log('LOGGED')
  next()
}
```

Notice the call above to next (). Calling this function invokes the next middleware function in the app.

The next () function is not a part of the Node.js or Express API, but is the third argument that is passed to the middleware function. The next () function could be named anything, but by convention it is always named “next”.

To avoid confusion, always use this convention.

To load the middleware function, call `app.use()`, specifying the middleware function. For example, the following code loads the `myLogger` middleware function before the route to the root path (`/`).

```
var express = require('express')
var app = express()

var myLogger = function (req, res, next) {
  console.log('LOGGED')
  next()
}

app.use(myLogger)

app.get('/', function (req, res) {
  res.send('Hello World!')
})

app.listen(3000)
```

Every time the app receives a request, it prints the message “LOGGED” to the terminal.

The order of middleware loading is important: middleware functions that are loaded first are also executed first.

If `myLogger` is loaded after the route to the root path, the request never reaches it and the app doesn't print “LOGGED”, because the route handler of the root path terminates the request-response cycle.

The middleware function `myLogger` simply prints a message, then passes on the request to the next middleware function in the stack by calling the `next()` function.

## Middleware function requestTime

Next, we'll create a middleware function called “requestTime” and add a property called `requestTime` to the request object.

```
var requestTime = function (req, res, next) {
  req.requestTime = Date.now()
  next()
}
```

The app now uses the `requestTime` middleware function. Also, the callback function of the root path route uses the property that the middleware function adds to `req` (the request object).

```
var express = require('express')
var app = express()

var requestTime = function (req, res, next) {
  req.requestTime = Date.now()
  next()
}

app.use(requestTime)

app.get('/', function (req, res) {
  var responseText = 'Hello World!<br>'
  responseText += '<small>Requested at: ' + req.requestTime + '</small>'
  res.send(responseText)
})

app.listen(3000)
```

When you make a request to the root of the app, the app now displays the timestamp of your request in the browser.

## Middleware function validateCookies

Finally, we'll create a middleware function that validates incoming cookies and sends a 400 response if cookies are invalid.

Here's an example function that validates cookies with an external async service.

```
async function cookieValidator (cookies) {
  try {
    await externallyValidateCookie(cookies.testCookie)
  } catch {
    throw new Error('Invalid cookies')
  }
}
```

Here we use the `cookie-parser` middleware to parse incoming cookies off the `req` object and pass them to our `cookieValidator` function. The `validateCookies` middleware returns a Promise that upon rejection will automatically trigger our error handler.

```
var express = require('express')
var cookieParser = require('cookie-parser')
var cookieValidator = require('./cookieValidator')

var app = express()

async function validateCookies (req, res, next) {
  await cookieValidator(req.cookies)
  next()
}

app.use(cookieParser())

app.use(validateCookies)

// error handler
app.use(function (err, req, res, next) {
  res.status(400).send(err.message)
})

app.listen(3000)
```

## Using middleware

Express is a routing and middleware web framework that has minimal functionality of its own: An Express application is essentially a series of middleware function calls.

**Middleware** functions are functions that have access to the [request object](#) (`req`), the [response object](#) (`res`), and the next middleware function in the application's request-response cycle. The next middleware function is commonly denoted by a variable named `next`.

Middleware functions can perform the following tasks:

- Execute any code.
- Make changes to the request and the response objects.
- End the request-response cycle.

- Call the next middleware function in the stack.

If the current middleware function does not end the request-response cycle, it must call `next()` to pass control to the next middleware function. Otherwise, the request will be left hanging.

An Express application can use the following types of middleware:

- [Application-level middleware](#)
- [Router-level middleware](#)
- [Error-handling middleware](#)
- [Built-in middleware](#)
- [Third-party middleware](#)

You can load application-level and router-level middleware with an optional mount path. You can also load a series of middleware functions together, which creates a sub-stack of the middleware system at a mount point.

## Application-level middleware

Bind application-level middleware to an instance of the [app object](#) by using the `app.use()` and `app.METHOD()` functions, where `METHOD` is the HTTP method of the request that the middleware function handles (such as GET, PUT, or POST) in lowercase.

This example shows a middleware function with no mount path. The function is executed every time the app receives a request.

```
var express = require('express')
var app = express()

app.use(function (req, res, next) {
  console.log('Time:', Date.now())
  next()
})
```

This example shows a middleware function mounted on the `/user/:id` path. The function is executed for any type of HTTP request on the `/user/:id` path.

```
app.use('/user/:id', function (req, res, next) {
  console.log('Request Type:', req.method)
  next()
})
```

This example shows a route and its handler function (middleware system). The function handles GET requests to the `/user/:id` path.

```
app.get('/user/:id', function (req, res, next) {
  res.send('USER')
})
```

Here is an example of loading a series of middleware functions at a mount point, with a mount path. It illustrates a middleware sub-stack that prints request info for any type of HTTP request to the `/user/:id` path.

```
app.use('/user/:id', function (req, res, next) {
  console.log('Request URL:', req.originalUrl)
  next()
}, function (req, res, next) {
  console.log('Request Type:', req.method)
  next()
})
```

Route handlers enable you to define multiple routes for a path. The example below defines two routes for GET requests to the `/user/:id` path. The second route will not cause any problems, but it will never get called because the first route ends the request-response cycle.

This example shows a middleware sub-stack that handles GET requests to the `/user/:id` path.

```
app.get('/user/:id', function (req, res, next) {
  console.log('ID:', req.params.id)
  next()
}, function (req, res, next) {
  res.send('User Info')
})

// handler for the /user/:id path, which prints the user ID
app.get('/user/:id', function (req, res, next) {
  res.end(req.params.id)
})
```

To skip the rest of the middleware functions from a router middleware stack, call `next('route')` to pass control to the next route. **NOTE:** `next('route')` will work only in middleware functions that were loaded by using the `app.METHOD()` or `router.METHOD()` functions.

This example shows a middleware sub-stack that handles GET requests to the `/user/:id` path.

```
app.get('/user/:id', function (req, res, next) {
```

```

// if the user ID is 0, skip to the next route
if (req.params.id === '0') next('route')
// otherwise pass the control to the next middleware function in this stack
else next()
}, function (req, res, next) {
  // send a regular response
  res.send('regular')
})

// handler for the /user/:id path, which sends a special response
app.get('/user/:id', function (req, res, next) {
  res.send('special')
})

```

Middleware can also be declared in an array for reusability.

This example shows an array with a middleware sub-stack that handles GET requests to the `/user/:id` path

```

function logOriginalUrl (req, res, next) {
  console.log('Request URL:', req.originalUrl)
  next()
}

function logMethod (req, res, next) {
  console.log('Request Type:', req.method)
  next()
}

var logStuff = [logOriginalUrl, logMethod]
app.get('/user/:id', logStuff, function (req, res, next) {
  res.send('User Info')
})

```

Middleware is an often misunderstood topic since it sounds and appears very complicated, but in reality middleware is actually really straightforward. The entire idea of middleware is to execute some code before the controller action that sends the response and after the server gets the request from the client. Essentially it is code that executes in the middle of your request, hence the name middleware. Before I get too in

depth on the details of middleware, though, I want to setup a basic Express server with two routes.

## What Is Middleware?

I talked briefly about middleware as functions that execute after the server receives the request and before the controller action sends the response, but there are a few more things that are specific to middleware. The biggest thing is that middleware functions have access to the response (`res`) and request (`req`) variables and can modify them or use them as needed. Middleware functions also have a third parameter which is a `next` function. This function is important since it must be called from a middleware for the next middleware to be executed. If this function is not called then none of the other middleware including the controller action will be called.

This is all a bit difficult to understand just from text so in the next section we are going to create a logging middleware that will log the url of the request a user makes.

## How To Create Logging Middleware

As I mentioned in the previous section, middleware takes three parameters, `req`, `res`, and `next`, so in order to create middleware we need to create a function that has those three inputs.

```
const express = require('express')
const app = express()

app.get('/', (req, res) => {
  res.send('Home Page')
})

app.get('/users', (req, res) => {
  res.send('Users Page')
})

function loggingMiddleware(req, res, next) { console.log('Inside Middleware')}
app.listen(3000, () => console.log('Server Started'))
```

We now have the shell of a basic middleware function defined with some placeholder content, but the application is not using it. Express has a few different ways you can define middleware to be used, but for this example we will make this middleware execute before every single controller action by adding it to the application level. This can be done by using the `use` function on the `app` variable like this.

```
const express = require('express')
const app = express()

app.use(loggingMiddleware)
app.get('/', (req, res) => {
  res.send('Home Page')
})
```



```
app.get('/users', (req, res) => {
  res.send('Users Page')
})

function loggingMiddleware(req, res, next) {
  console.log('Inside Middleware')
}
```

```
app.listen(3000, () => console.log('Server Started'))
```

The application is now using the middleware that we defined and if we restart our server and navigate to any of the pages in our app you will notice that in the console the message **Inside Middleware** appears. This is great, but there is a slight problem. The application now loads forever and never actually finishes the request. This is because in our middleware we are not calling the `next` function so the controller action never gets called. We can fix this by calling `next` after our logging.

```
const express = require('express')
const app = express()

app.use(loggingMiddleware)

app.get('/', (req, res) => {
  res.send('Home Page')
})

app.get('/users', (req, res) => {
  res.send('Users Page')
})

function loggingMiddleware(req, res, next) {
  console.log('Inside Middleware')
  next()}

app.listen(3000, () => console.log('Server Started'))
```

Now if you restart the server you will notice that everything is logging correctly, and the web page is properly loading. The next thing to do is to actually log out the URL that the user is accessing inside the middleware. This is where the `req` variable will come in handy.

```
const express = require('express')
const app = express()

app.use(loggingMiddleware)

app.get('/', (req, res) => {
  res.send('Home Page')
})

app.get('/users', (req, res) => {
  res.send('Users Page')
```

```

})

function loggingMiddleware(req, res, next) {
  console.log(` ${new Date().toISOString()}: ${req.originalUrl}`) next()
}

app.listen(3000, () => console.log('Server Started'))

```

The logging middleware is now working 100% correctly on all the routes in the application, but we have only scratched the surface on the usefulness of middleware. In the next example we are going to take a look at creating a simple authorization middleware for the users page.

## Advanced Middleware Example

To get started we need to create another function to use as middleware.

```

const express = require('express')
const app = express()

app.use(loggingMiddleware)

app.get('/', (req, res) => {
  res.send('Home Page')
})

app.get('/users', (req, res) => {
  res.send('Users Page')
})

function loggingMiddleware(req, res, next) {
  console.log(` ${new Date().toISOString()}: ${req.originalUrl}`)
  next()
}

function authorizeUsersAccess(req, res, next) { console.log('authorizeUsersAccess Middleware') next()}
app.listen(3000, () => console.log('Server Started'))

```

This is just a shell of a function to be used as middleware, but we can add it to our users page route now in order to ensure that our middleware is only being executed on the users page route. This can be done by adding the function as a parameter to the `app.get` function for the users page.

```

const express = require('express')
const app = express()

app.use(loggingMiddleware)

app.get('/', (req, res) => {
  res.send('Home Page')
})

app.get('/users', authorizeUsersAccess, (req, res) => { res.send('Users Page')

```

```

})

function loggingMiddleware(req, res, next) {
  console.log(` ${new Date().toISOString()}: ${req.originalUrl}`)
  next()
}

function authorizeUsersAccess(req, res, next) {
  console.log('authorizeUsersAccess Middleware')
  next()
}

app.listen(3000, () => console.log('Server Started'))

```

Now if you restart the server and go to the users page you should see the message **authorizeUsersAccess Middleware**, but if you go to the home page this message will not show up. We now have middleware that only executes on a single route in the application. The next thing to do is fill in the logic of this function so that if the user does not have access to the page they will get an error message instead.

```

const express = require('express')
const app = express()

app.use(loggingMiddleware)

app.get('/', (req, res) => {
  res.send('Home Page')
})

app.get('/users', authorizeUsersAccess, (req, res) => {
  res.send('Users Page')
})

function loggingMiddleware(req, res, next) {
  console.log(` ${new Date().toISOString()}: ${req.originalUrl}`)
  next()
}

function authorizeUsersAccess(req, res, next) {
  if (req.query.admin === 'true') { next() } else { res.send('ERROR: You must be an admin') }}

app.listen(3000, () => console.log('Server Started'))

```

This middleware now checks to see if the query parameter `admin=true` is in the URL and if it is not an error message is shown to the user. You can test this by going to `http://localhost:3000/users` and you will see an error message explaining that you are not a admin. If you instead go to `http://localhost:3000/users?admin=true` you will be sent the normal users page since you set the query parameter of admin to true.

One other thing that is really useful with middleware is the ability to send data between middleware. There is no way to do this with the next function, but you can modify the `req` or `res` variables to set your own custom data. For example in the previous

example if we wanted to set a variable to true if the user was a admin we could easily do that.

```
const express = require('express')
const app = express()

app.use(loggingMiddleware)

app.get('/', (req, res) => {
  res.send('Home Page')
})

app.get('/users', authorizeUsersAccess, (req, res) => {
  console.log(req.admin)  res.send('Users Page')
})

function loggingMiddleware(req, res, next) {
  console.log(`${new Date().toISOString()}: ${req.originalUrl}`)
  next()
}

function authorizeUsersAccess(req, res, next) {
  if (req.query.admin === 'true') {
    req.admin = true    next()
  } else {
    res.send('ERROR: You must be an admin')
  }
}

app.listen(3000, () => console.log('Server Started'))
```

This code sets an admin variable on the `req` object which is then accessed in the controller action for the users page.

## Middleware Additional Information

This is the majority of everything you need to know about middleware functions, but there a few extra things that are important to know.

### 1. Controller Actions Are Just Like Middleware

One thing you may have noticed is that controller actions which have a `req`, and `res` variable are very similar to middleware. That is because they are essentially middleware, but with no other middleware that comes after them. They are the end of the chain which is why there are never any next calls inside the controller action.

## 2. Calling next Is Not The Same As Calling return

By far the biggest mistake I see developers make when working with middleware is that they treat the `next` function as if it exited out of the middleware. Take for example this middleware.

```
function middleware(req, res, next) {
  if (req.valid) {
    next()
  }
  res.send('Invalid Request')
}
```

At face value this code looks correct. If the request is valid then the `next` function is called and if it isn't valid then it is sending an error message. The problem is that the `next` function does not actually return from the middleware function. This means that when `next` is called the next middleware will execute and that will continue until no more middleware is left to execute. Then after all the middleware after this middleware is done executing the code will pick back up right after the `next` call in each of the middleware. That means that in this middleware the error message will always be sent to the user which is obviously not what you want. An easy way to prevent this is by simply returning when you call `next`

```
function middleware(req, res, next) {
  if (req.valid) {
    return next() }
  res.send('Invalid Request')
}
```

Now the code will no longer execute after calling `next` since it will return out of the function. An easy way to see this issue in action is with the following code.

```
const express = require('express')
const app = express()

app.get('/', middleware, (req, res) => {
  console.log('Inside Home Page')
  res.send('Home Page')
})

function middleware(req, res, next) {
  console.log('Before Next')
  next()
  console.log('After Next')
}

app.listen(3000, () => console.log('Server Started'))
```

When you run this code and go to the home page the console will print out the following messages in order.

Before Next  
Inside Home Page  
After Next

Essentially what is happening is the middleware is called and it logs out the before statement. Then next is called so the next set of middleware is called which is the controller action where the home page message is logged. Lastly the controller action finishes executing so the middleware then executes the code after `next` which logs out the after statement.

### 3. Middleware Will Execute In Order

This may seem self-explanatory but when you define middleware it will execute in the order it is used. Take for example the following code.

```
const express = require('express')
const app = express()

app.use(middlewareThree)
app.use(middlewareOne)

app.get('/', middlewareTwo, middlewareFour, (req, res) => {
  console.log('Inside Home Page')
  res.send('Home Page')
})

function middlewareOne(req, res, next) {
  console.log('Middleware One')
  next()
}

function middlewareTwo(req, res, next) {
  console.log('Middleware Two')
  next()
}

function middlewareThree(req, res, next) {
  console.log('Middleware Three')
  next()
}

function middlewareFour(req, res, next) {
  console.log('Middleware Four')
  next()
}

app.listen(3000, () => console.log('Server Started'))
```

Since the `app.use` statements come first the middleware in those statements will be executed first in the order they were added. Next the `app.get` middleware is defined and again they will be executed in the order they are in the `app.get` function. This will lead to the following console output if ran.

Middleware Three  
Middleware One  
Middleware Two  
Middleware Four

## Conclusion

That is all there is to know about middleware. Middleware is incredibly powerful for cleaning up code and making things like user authorization and authentication much easier, but it can be used for so much more than just that because of the incredible flexibility of middleware.

# Router

A `router` object is an isolated instance of middleware and routes. You can think of it as a “mini-application,” capable only of performing middleware and routing functions. Every Express application has a built-in app router.

A router behaves like middleware itself, so you can use it as an argument to [app.use\(\)](#) or as the argument to another router’s [use\(\)](#) method.

The top-level `express` object has a [Router\(\)](#) method that creates a new `router` object.

## express.Router

Use the `express.Router` class to create modular, mountable route handlers. A `Router` instance is a complete middleware and routing system; for this reason, it is often referred to as a “mini-app”.

The following example creates a router as a module, loads a middleware function in it, defines some routes, and mounts the router module on a path in the main app.

Create a router file named `birds.js` in the app directory, with the following content:

```
var express = require('express')
var router = express.Router()

// middleware that is specific to this router
router.use(function timeLog (req, res, next) {
  console.log('Time: ', Date.now())
  next()
})
// define the home page route
```

```
router.get('/', function (req, res) {  
  res.send('Birds home page')  
})  
// define the about route  
router.get('/about', function (req, res) {  
  res.send('About birds')  
})  
  
module.exports = router
```

Then, load the router module in the app:

```
var birds = require('./birds')  
  
// ...  
  
app.use('/birds', birds)
```

The app will now be able to handle requests to `/birds` and `/birds/about`, as well as call the `timeLog` middleware function that is specific to the route.

## Events Module

Node.js has a built-in module, called "Events", where you can create-, fire-, and listen for- your own events.

To include the built-in Events module use the `require()` method. In addition, all event properties and methods are an instance of an EventEmitter object. To be able to access these properties and methods, create an EventEmitter object:

## 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('I hear a scream!');
}

//Assign the event handler to an event:
eventEmitter.on('scream', myEventHandler);

//Fire the 'scream' event:
eventEmitter.emit('scream');
```