

# What is Node.js

Node.js is a cross-platform runtime environment and library for running JavaScript applications outside the browser. It is used for creating server-side and networking web applications. It is open source and free to use. It can be downloaded from this link <https://nodejs.org/en/>

Many of the basic modules of Node.js are written in JavaScript. Node.js is mostly used to run real-time server applications.

The definition given by its official documentation is as follows:

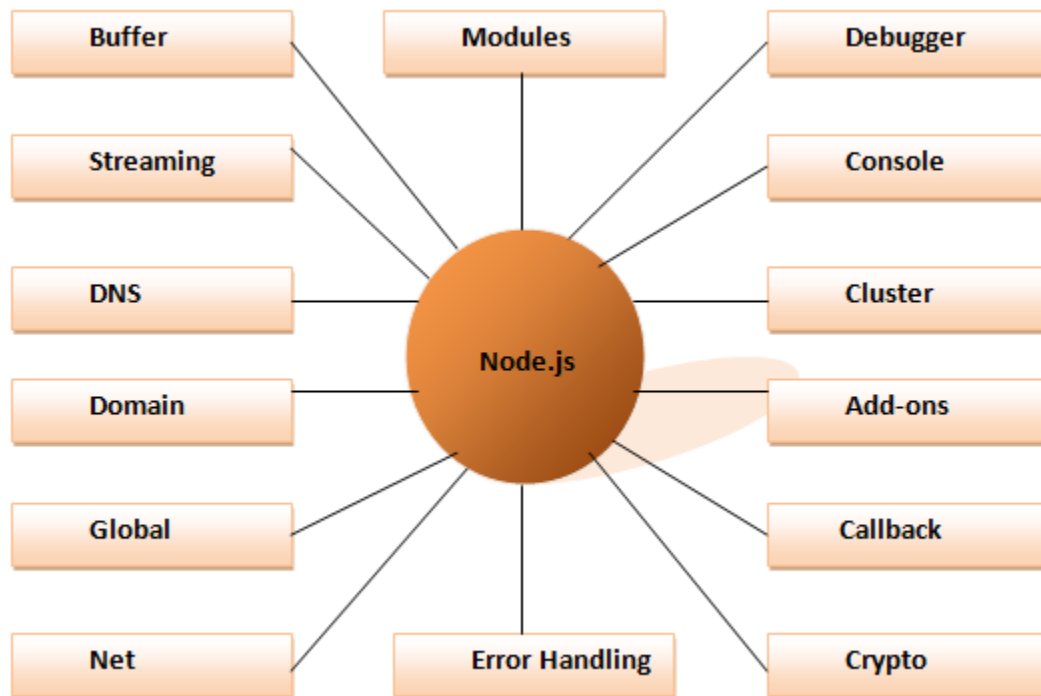
?Node.js is a platform built on Chrome's JavaScript runtime for easily building fast and scalable network applications. Node.js uses an event-driven, non-blocking I/O model that makes it lightweight and efficient, perfect for data-intensive real-time applications that run across distributed devices.?

Node.js also provides a rich library of various JavaScript modules to simplify the development of web applications.

1. **Node.js** = **Runtime** Environment + JavaScript Library

## Different parts of Node.js

The following diagram specifies some important parts of Node.js:



# Features of Node.js

Following is a list of some important features of Node.js that makes it the first choice of software architects.

1. **Extremely fast:** Node.js is built on Google Chrome's V8 JavaScript Engine, so its library is very fast in code execution.
2. **I/O is Asynchronous and Event Driven:** All APIs of Node.js library are asynchronous i.e. non-blocking. So a Node.js based server never waits for an API to return data. The server moves to the next API after calling it and a notification mechanism of Events of Node.js helps the server to get a response from the previous API call. It is also a reason that it is very fast.
3. **Single threaded:** Node.js follows a single threaded model with event looping.
4. **Highly Scalable:** Node.js is highly scalable because event mechanism helps the server to respond in a non-blocking way.
5. **No buffering:** Node.js cuts down the overall processing time while uploading audio and video files. Node.js applications never buffer any data. These applications simply output the data in chunks.
6. **Open source:** Node.js has an open source community which has produced many excellent modules to add additional capabilities to Node.js applications.
7. **License:** Node.js is released under the MIT license.

# Download Node.js

The official Node.js website has installation instructions for Node.js: <https://nodejs.org>

## Getting Started

Once you have downloaded and installed Node.js on your computer, let's try to display "Hello World" in a web browser.

Create a Node.js file named "myfirst.js", and add the following code:

myfirst.js

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

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

Save the file on your computer: C:\Users\Your Name\myfirst.js

The code tells the computer to write "Hello World!" if anyone (e.g. a web browser) tries to access your computer on port 8080.

For now, you do not have to understand the code. It will be explained later.

## Command Line Interface

Node.js files must be initiated in the "Command Line Interface" program of your computer.

How to open the command line interface on your computer depends on the operating system. For Windows users, press the start button and look for "Command Prompt", or simply write "cmd" in the search field.

Navigate to the folder that contains the file "myfirst.js", the command line interface window should look something like this:

```
C:\Users\Your Name>_
```

## Initiate the Node.js File

The file you have just created must be initiated by Node.js before any action can take place.

Start your command line interface, write `node myfirst.js` and hit enter:

```
Initiate "myfirst.js":
```

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

Now, your computer works as a server!

If anyone tries to access your computer on port 8080, they will get a "Hello World!" message in return!

Start your internet browser, and type in the address: `http://localhost:8080`

# What is a Module in Node.js?

Consider modules to be the same as JavaScript libraries.

A set of functions you want to include in your application.

## Built-in Modules

Node.js has a set of built-in modules which you can use without any further installation.

Look at our Built-in Modules Reference for a complete list of modules.

## Include Modules

To include a module, use the `require()` function with the name of the module:

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

Now your application has access to the HTTP module, and is able to create a server:

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

## Create Your Own Modules

You can create your own modules, and easily include them in your applications.

The following example creates a module that returns a date and time object:

## Example

Create a module that returns the current date and time:

```
exports.myDateTime = function () {  
  return Date();  
};
```

Use the `exports` keyword to make properties and methods available outside the module file.

Save the code above in a file called "myfirstmodule.js"

## Include Your Own Module

Now you can include and use the module in any of your Node.js files.

## Example

Use the module "myfirstmodule" in a Node.js file:

```
var http = require('http');  
var dt = require('./myfirstmodule');  
  
http.createServer(function (req, res) {  
  res.writeHead(200, {'Content-Type': 'text/html'});  
  res.write("The date and time are currently: " + dt.myDateTime());  
  res.end();  
}).listen(8080);
```

Notice that we use `./` to locate the module, that means that the module is located in the same folder as the Node.js file.

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

Initiate demo\_module.js:

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

If you have followed the same steps on your computer, you will see the same result as the example: <http://localhost:8080>

# Node.js console-based Example

*File: console\_example1.js*

1. `console.log('Hello world');`

Open Node.js command prompt and run the following code:

1. `node console_example1.js`  
Here, `console.log()` function displays message on console.



# The Built-in 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).

To include the HTTP module, use the `require()` method:

```
var http = require('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.

Use the `createServer()` method to create an HTTP server:

### 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.

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

Initiate demo\_http.js:

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

If you have followed the same steps on your computer, you will see the same result as the example: `http://localhost:8080`

# 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.

# 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\Your Name>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/summer

Will produce this result:

```
/summer
```

http://localhost:8080/winter

Will produce this result:

```
/winter
```

## 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:

Initiate demo\_querystring.js:

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

The address:

<http://localhost:8080/?year=2017&month=July>

Will produce this result:

2017 July

# Node.js as a File Server

The Node.js file system module allows you to work with the file system on your computer.

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

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

Common use for the File System module:

- Read files
- Create files
- Update files
- Delete files
- Rename files

## Read Files

The `fs.readFile()` method is used to read files on your computer.

Assume we have the following HTML file (located in the same folder as Node.js):

demofile1.html

```
<html>
<body>
<h1>My Header</h1>
<p>My paragraph.</p>
</body>
</html>
```

Create a Node.js file that reads the HTML file, and return the content:

### Example

```
var http = require('http');
var fs = require('fs');
http.createServer(function (req, res) {
  fs.readFile('demofile1.html', function(err, data) {
```

```
    res.writeHead(200, {'Content-Type': 'text/html'});  
    res.write(data);  
    return res.end();  
  });  
}).listen(8080);
```

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

Initiate demo\_readfile.js:

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

If you have followed the same steps on your computer, you will see the same result as the example: <http://localhost:8080>

## Create Files

The File System module has methods for creating new files:

- `fs.appendFile()`
- `fs.open()`
- `fs.writeFile()`

The `fs.appendFile()` method appends specified content to a file. If the file does not exist, the file will be created:

### Example

Create a new file using the `appendFile()` method:

```
var fs = require('fs');  
  
fs.appendFile('mynewfile1.txt', 'Hello content!', function (err) {  
  if (err) throw err;  
  console.log('Saved!');  
});
```

The `fs.open()` method takes a "flag" as the second argument, if the flag is "w" for "writing", the specified file is opened for writing. If the file does not exist, an empty file is created:

## Example

Create a new, empty file using the `open()` method:

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

fs.open('mynewfile2.txt', 'w', function (err, file) {
  if (err) throw err;
  console.log('Saved!');
});
```

The `fs.writeFile()` method replaces the specified file and content if it exists. If the file does not exist, a new file, containing the specified content, will be created:

## Example

Create a new file using the `writeFile()` method:

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

fs.writeFile('mynewfile3.txt', 'Hello content!', function (err) {
  if (err) throw err;
  console.log('Saved!');
});
```

# Update Files

The File System module has methods for updating files:

- `fs.appendFile()`
- `fs.writeFile()`

The `fs.appendFile()` method appends the specified content at the end of the specified file:

## Example

Append "This is my text." to the end of the file "mynewfile1.txt":

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

fs.appendFile('mynewfile1.txt', ' This is my text.', function (err)
{
  if (err) throw err;
  console.log('Updated!');
});
```

The `fs.writeFile()` method replaces the specified file and content:

## Example

Replace the content of the file "mynewfile3.txt":

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

fs.writeFile('mynewfile3.txt', 'This is my text', function (err) {
  if (err) throw err;
  console.log('Replaced!');
});
```

# Delete Files

To delete a file with the File System module, use the `fs.unlink()` method.

The `fs.unlink()` method deletes the specified file:

## Example

Delete "mynewfile2.txt":

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

fs.unlink('mynewfile2.txt', function (err) {
  if (err) throw err;
});
```



```
    console.log('File deleted!');  
  });
```

## Rename Files

To rename a file with the File System module, use the `fs.rename()` method.

The `fs.rename()` method renames the specified file:

### Example

Rename "mynewfile1.txt" to "myrenamedfile.txt":

```
var fs = require('fs');  
  
fs.rename('mynewfile1.txt', 'myrenamedfile.txt', function (err) {  
  if (err) throw err;  
  console.log('File Renamed!');  
});
```

## Upload Files

You can also use Node.js to upload files to your computer.

Read how in our Node.js Upload Files chapter.



# The Built-in URL Module

The URL module splits up a web address into readable parts.

To include the URL module, use the `require()` method:

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

Parse an address with the `url.parse()` method, and it will return a URL object with each part of the address as properties:

## Example

Split a web address into readable parts:

```
var url = require('url');
var adr
= 'http://localhost:8080/default.htm?year=2017&month=february';
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=2017&month=february'

var qdata = q.query; //returns an object: { year: 2017, month:
'february' }
console.log(qdata.month); //returns 'february'
```

## Node.js File Server

Now we know how to parse the query string, and in the previous chapter we learned how to make Node.js behave as a file server. Let us combine the two, and serve the file requested by the client.

Create two html files and save them in the same folder as your node.js files.

```
summer.html
```

```
<!DOCTYPE html>
<html>
<body>
<h1>Summer</h1>
<p>I love the sun!</p>
</body>
</html>
```

winter.html

```
<!DOCTYPE html>
<html>
<body>
<h1>Winter</h1>
<p>I love the snow!</p>
</body>
</html>
```

Create a Node.js file that opens the requested file and returns the content to the client. If anything goes wrong, throw a 404 error:

demo\_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);
```

Remember to initiate the file:

```
Initiate demo_fileserver.js:
```

```
C:\Users\Your Name>node demo_fileserver.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/summer.html>

Will produce this result:

# Summer

I love the sun!

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

Will produce this result:

# Winter

I love the snow!

# Events in Node.js

Every action on a computer is an event. Like when a connection is made or a file is opened.

Objects in Node.js can fire events, like the `readStream` object fires events when opening and closing a file:

## Example

```
var fs = require('fs');
var rs = fs.createReadStream('./demofile.txt');
rs.on('open', function () {
  console.log('The file is open');
});
```

## 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:

```
var events = require('events');
var EventEmitter = new events.EventEmitter();
```

## 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.

## Example

```
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');
```

# The Formidable Module

There is a very good module for working with file uploads, called "Formidable".

The Formidable module can be downloaded and installed using NPM:

```
C:\Users\Your Name>npm install formidable
```

After you have downloaded the Formidable module, you can include the module in any application:

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

## Upload Files

Now you are ready to make a web page in Node.js that lets the user upload files to your computer:

### Step 1: Create an Upload Form

Create a Node.js file that writes an HTML form, with an upload field:

#### Example

This code will produce an HTML form:

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

http.createServer(function (req, res) {
  res.writeHead(200, {'Content-Type': 'text/html'});
  res.write('<form action="fileupload" method="post"
  enctype="multipart/form-data">');
  res.write('<input type="file" name="filetoupload"><br>');
  res.write('<input type="submit">');
  res.write('</form>');
  return res.end();
}).listen(8080);
```



## Step 2: Parse the Uploaded File

Include the Formidable module to be able to parse the uploaded file once it reaches the server.

When the file is uploaded and parsed, it gets placed on a temporary folder on your computer.

### Example

The file will be uploaded, and placed on a temporary folder:

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

http.createServer(function (req, res) {
  if (req.url == '/fileupload') {
    var form = new formidable.IncomingForm();
    form.parse(req, function (err, fields, files) {
      res.write('File uploaded');
      res.end();
    });
  } else {
    res.writeHead(200, {'Content-Type': 'text/html'});
    res.write('<form action="fileupload" method="post"
    enctype="multipart/form-data">');
    res.write('<input type="file" name="filetoupload"><br>');
    res.write('<input type="submit">');
    res.write('</form>');
    return res.end();
  }
}).listen(8080);
```

## Step 3: Save the File

When a file is successfully uploaded to the server, it is placed on a temporary folder.

The path to this directory can be found in the "files" object, passed as the third argument in the `parse()` method's callback function.

To move the file to the folder of your choice, use the File System module, and rename the file:

## Example

Include the fs module, and move the file to the current folder:

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

http.createServer(function (req, res) {
  if (req.url == '/fileupload') {
    var form = new formidable.IncomingForm();
    form.parse(req, function (err, fields, files) {
      var oldpath = files.fileupload.path;
      var newpath = 'C:/Users/Your Name/' + files.fileupload.name;
      fs.rename(oldpath, newpath, function (err) {
        if (err) throw err;
        res.write('File uploaded and moved!');
        res.end();
      });
    });
  } else {
    res.writeHead(200, {'Content-Type': 'text/html'});
    res.write('<form action="fileupload" method="post"
enctype="multipart/form-data">');
    res.write('<input type="file" name="fileupload"><br>');
    res.write('<input type="submit">');
    res.write('</form>');
    return res.end();
  }
}).listen(8080);
```

# Express.js Tutorial

Express.js tutorial provides basic and advanced concepts of Express.js. Our Express.js tutorial is designed for beginners and professionals both.

Express.js is a web framework for Node.js. It is a fast, robust and asynchronous in nature.

Our Express.js tutorial includes all topics of Express.js such as Express.js installation on windows and linux, request object, response object, get method, post method, cookie management, scaffolding, file upload, template etc.

## What is Express.js

Express is a fast, assertive, essential and moderate web framework of Node.js. You can assume express as a layer built on the top of the Node.js that helps manage a server and routes. It provides a robust set of features to develop web and mobile applications.

Let's see some of the core features of Express framework:

- It can be used to design single-page, multi-page and hybrid web applications.
- It allows to setup middlewares to respond to HTTP Requests.
- It defines a routing table which is used to perform different actions based on HTTP method and URL.
- It allows to dynamically render HTML Pages based on passing arguments to templates.

---

## Why use Express

- Ultra fast I/O
- Asynchronous and single threaded
- MVC like structure
- Robust API makes routing easy

## How does Express look like

Let's see a basic Express.js app.

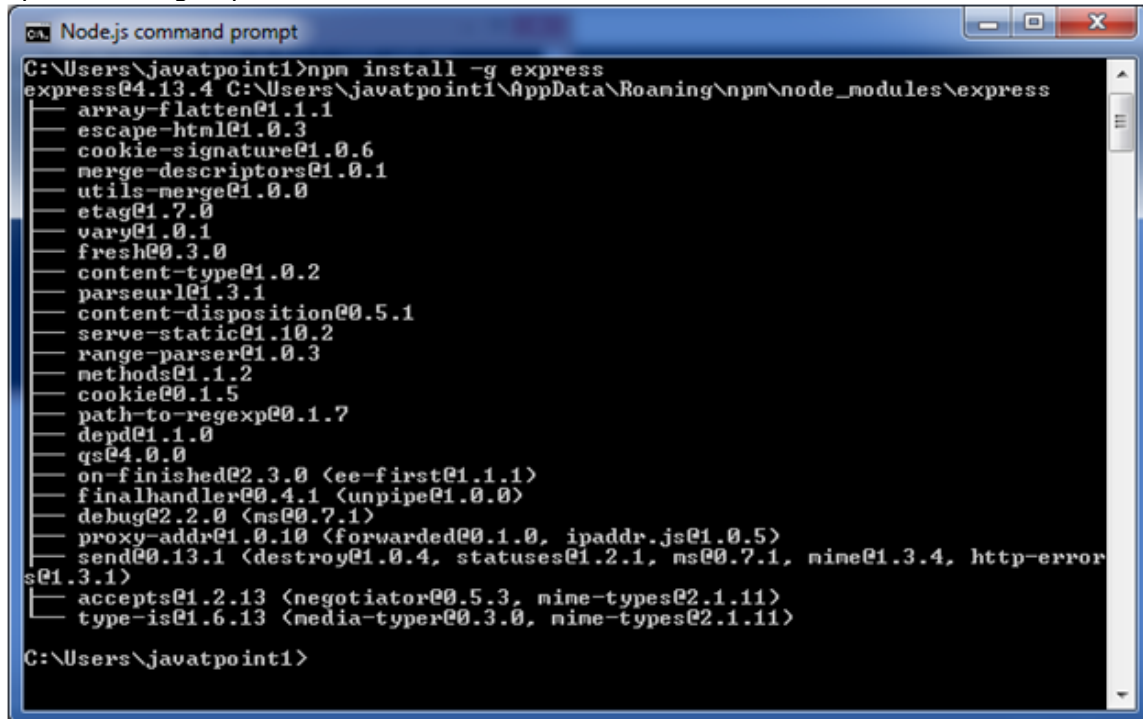
**File: basic\_express.js**

```
1. var express = require('express');
2. var app = express();
3. app.get('/', function (req, res) {
4.   res.send('Welcome to JavaTpoint!');
5. });
6. var server = app.listen(8000, function () {
7.   var host = server.address().address;
8.   var port = server.address().port;
9.   console.log('Example app listening at http://%s:%s', host, port);
10. });
```

# Install Express.js

Firstly, you have to install the express framework globally to create web application using Node terminal. Use the following command to install express framework globally.

1. `npm install -g express`



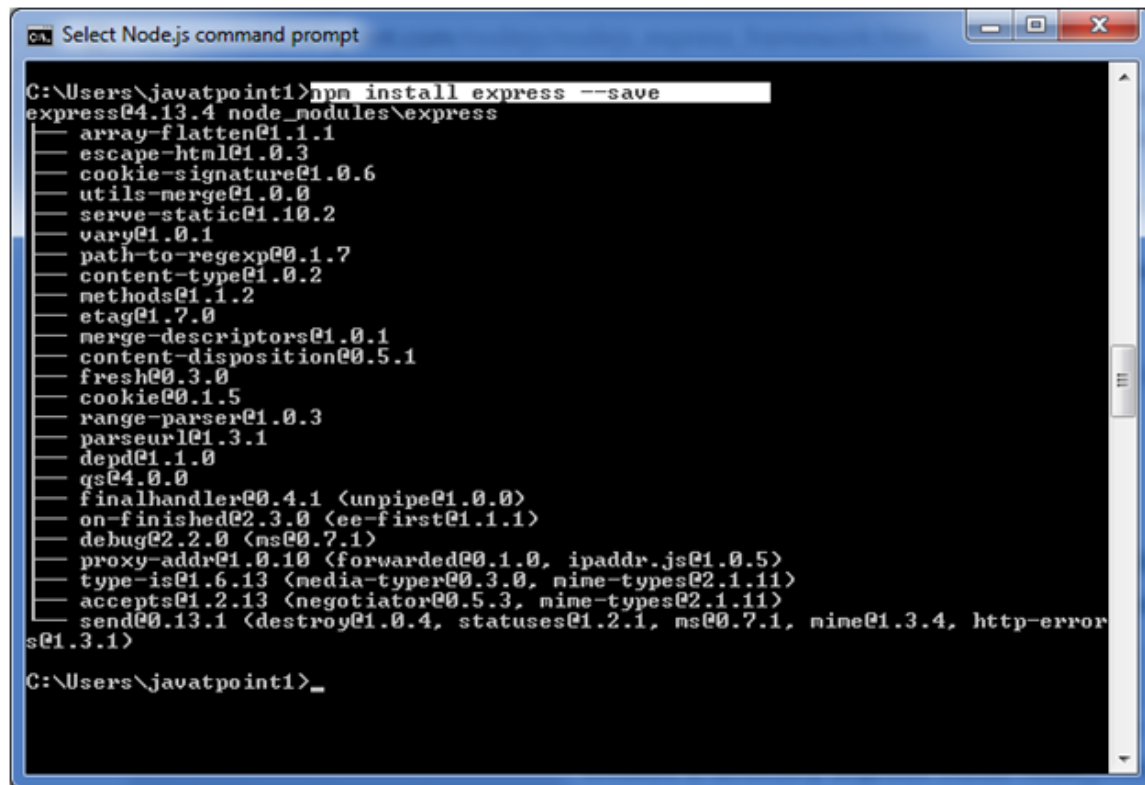
```
Node.js command prompt
C:\Users\javatpoint1>npm install -g express
express@4.13.4 C:\Users\javatpoint1\AppData\Roaming\npm\node_modules\express
├── array-flatten@1.1.1
├── escape-html@1.0.3
├── cookie-signature@1.0.6
├── merge-descriptors@1.0.1
├── utils-merge@1.0.0
├── etag@1.7.0
├── vary@1.0.1
├── fresh@0.3.0
├── content-type@1.0.2
├── parseurl@1.3.1
├── content-disposition@0.5.1
├── serve-static@1.10.2
├── range-parser@1.0.3
├── methods@1.1.2
├── cookie@0.1.5
├── path-to-regexp@0.1.7
├── depd@1.1.0
├── qs@4.0.0
├── on-finished@2.3.0 <ee-first@1.1.1>
├── finalhandler@0.4.1 <unpipe@1.0.0>
├── debug@2.2.0 <ms@0.7.1>
├── proxy-addr@1.0.10 <forwarded@0.1.0, ipaddr.js@1.0.5>
├── send@0.13.1 <destroy@1.0.4, statuses@1.2.1, ms@0.7.1, mime@1.3.4, http-error
s@1.3.1>
├── accepts@1.2.13 <negotiator@0.5.3, mime-types@2.1.11>
└── type-is@1.6.13 <media-typer@0.3.0, mime-types@2.1.11>

C:\Users\javatpoint1>
```

## Installing Express

Use the following command to install express:

1. `npm install express --save`

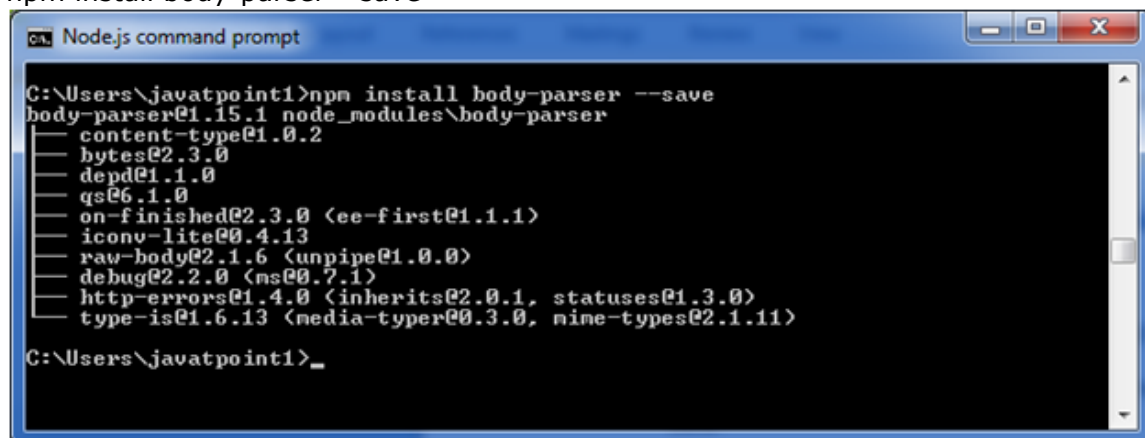


```
C:\Users\javatpoint1>npm install express --save
express@4.13.4 node_modules\express
├── array-flatten@1.1.1
├── escape-html@1.0.3
├── cookie-signature@1.0.6
├── utils-merge@1.0.0
├── serve-static@1.10.2
├── vary@1.0.1
├── path-to-regexp@0.1.7
├── content-type@1.0.2
├── methods@1.1.2
├── etag@1.7.0
├── merge-descriptors@1.0.1
├── content-disposition@0.5.1
├── fresh@0.3.0
├── cookie@0.1.5
├── range-parser@1.0.3
├── parseurl@1.3.1
├── depd@1.1.0
├── qs@4.0.0
├── finalhandler@0.4.1 <unpipe@1.0.0>
├── on-finished@2.3.0 <ee-first@1.1.1>
├── debug@2.2.0 <ms@0.7.1>
├── proxy-addr@1.0.10 <forwarded@0.1.0, ipaddr.js@1.0.5>
├── type-is@1.6.13 <media-typer@0.3.0, mime-types@2.1.11>
├── accepts@1.2.13 <negotiator@0.5.3, mime-types@2.1.11>
└── send@0.13.1 <destroy@1.0.4, statuses@1.2.1, ms@0.7.1, mime@1.3.4, http-error
s@1.3.1>
C:\Users\javatpoint1>_
```

The above command install express in node\_module directory and create a directory named express inside the node\_module. You should install some other important modules along with express. Following is the list:

- **body-parser:** This is a node.js middleware for handling JSON, Raw, Text and URL encoded form data.
- **cookie-parser:** It is used to parse Cookie header and populate req.cookies with an object keyed by the cookie names.
- **multer:** This is a node.js middleware for handling multipart/form-data.

1. npm install body-parser --save



```
C:\Users\javatpoint1>npm install body-parser --save
body-parser@1.15.1 node_modules\body-parser
├── content-type@1.0.2
├── bytes@2.3.0
├── depd@1.1.0
├── qs@6.1.0
├── on-finished@2.3.0 <ee-first@1.1.1>
├── iconv-lite@0.4.13
├── raw-body@2.1.6 <unpipe@1.0.0>
├── debug@2.2.0 <ms@0.7.1>
├── http-errors@1.4.0 <inherits@2.0.1, statuses@1.3.0>
└── type-is@1.6.13 <media-typer@0.3.0, mime-types@2.1.11>
C:\Users\javatpoint1>_
```

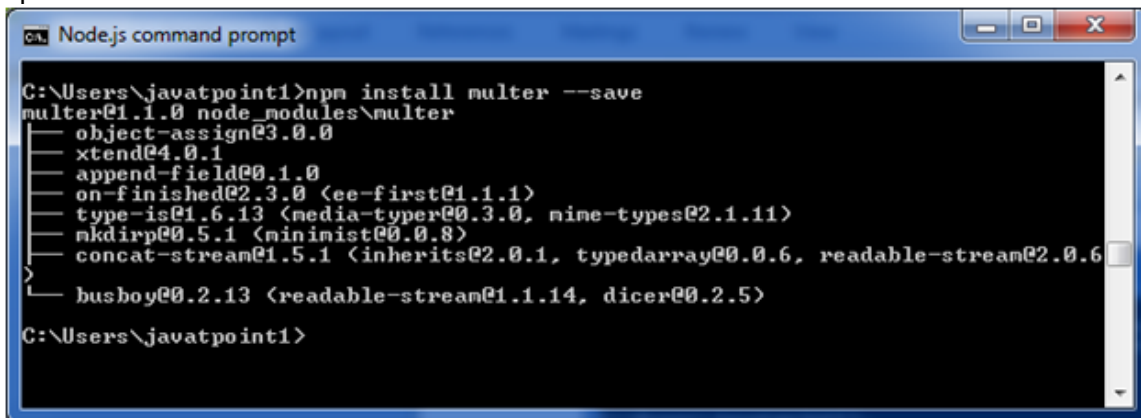
1. npm install cookie-parser --save



```
C:\Users\javatpoint1>npm install cookie-parser --save
cookie-parser@1.4.2 node_modules\cookie-parser
├── cookie-signature@1.0.6
└── cookie@0.2.4

C:\Users\javatpoint1>
```

1. npm install multer --save



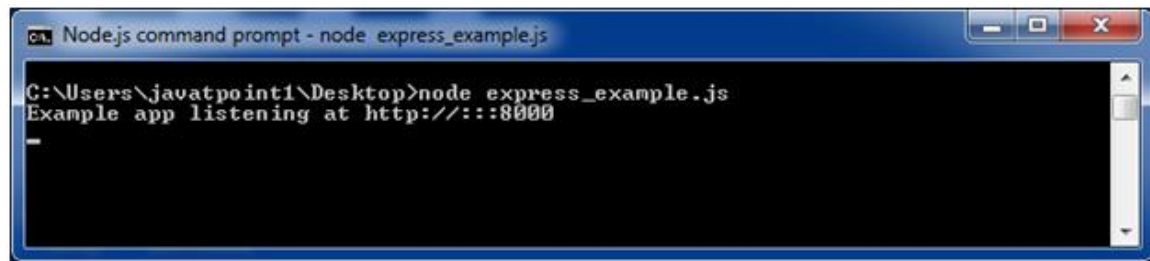
```
C:\Users\javatpoint1>npm install multer --save
multer@1.1.0 node_modules\multer
├── object-assign@3.0.0
├── xtend@4.0.1
├── append-field@0.1.0
├── on-finished@2.3.0 <ee-first@1.1.1>
├── type-is@1.6.13 <media-typer@0.3.0, mime-types@2.1.11>
├── mkdirp@0.5.1 <minimist@0.0.8>
├── concat-stream@1.5.1 <inherits@2.0.1, typedarray@0.0.6, readable-stream@2.0.6>
└── busboy@0.2.13 <readable-stream@1.1.14, dicer@0.2.5>

C:\Users\javatpoint1>
```

## Express.js App Example

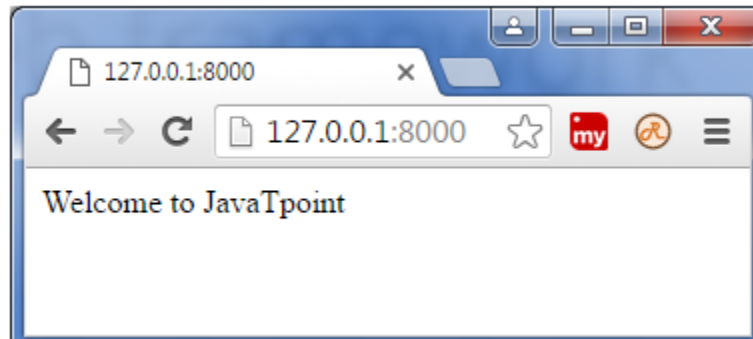
Let's take a simple Express app example which starts a server and listen on a local port. It only responds to homepage. For every other path, it will respond with a 404 Not Found error.

1. File: express\_example.js
1. var **express** = **require**('express');
2. var **app** = **express**();
3. app.get('/', function (req, res) {
4.   res.send('Welcome to JavaTpoint');
5. })
6. var **server** = **app**.listen(8000, function () {
7.   var **host** = **server**.address().address
8.   var **port** = **server**.address().port
9.   console.log("Example app listening at http://%s:%s", host, port)
10. })



```
Node.js command prompt - node express_example.js  
C:\Users\javatpoint1\Desktop>node express_example.js  
Example app listening at http://:::8000  
-
```

Open <http://127.0.0.1:8000/> in your browser to see the result.





# Express.js Request Object

Express.js Request and Response objects are the parameters of the callback function which is used in Express applications.

The express.js request object represents the HTTP request and has properties for the request query string, parameters, body, HTTP headers, and so on.

## Syntax:

1. `app.get('/', function (req, res) {`
  2.  `// --`
  3. `})`
- 

## Express.js Request Object Properties

The following table specifies some of the properties associated with request object.

Index	Properties	Description
1.	<code>req.app</code>	This is used to hold a reference to the instance of the express application that is using the middleware.
2.	<code>req.baseUrl</code>	It specifies the URL path on which a router instance was mounted.
3.	<code>req.body</code>	It contains key-value pairs of data submitted in the request body. By default, it is undefined, and is populated when you use body-parsing middleware such as <code>body-parser</code> .
4.	<code>req.cookies</code>	When we use <code>cookie-parser</code> middleware, this property is an object that contains cookies sent by the request.
5.	<code>req.fresh</code>	It specifies that the request is "fresh." it is the opposite of <code>req.stale</code> .
6.	<code>req.hostname</code>	It contains the hostname from the "host" http header.
7.	<code>req.ip</code>	It specifies the remote IP address of the request.

8.	req.ips	When the trust proxy setting is true, this property contains an array of IP addresses specified in the ?x-forwarded-for? request header.
9.	req.originalurl	This property is much like req.url; however, it retains the original request URL, allowing you to rewrite req.url freely for internal routing purposes.
10.	req.params	An object containing properties mapped to the named route ?parameters?. For example, if you have the route /user/:name, then the "name" property is available as req.params.name. This object defaults to {}.
11.	req.path	It contains the path part of the request URL.
12.	req.protocol	The request protocol string, "http" or "https" when requested with TLS.
13.	req.query	An object containing a property for each query string parameter in the route.
14.	req.route	The currently-matched route, a string.
15.	req.secure	A Boolean that is true if a TLS connection is established.
16.	req.signedcookies	When using cookie-parser middleware, this property contains signed cookies sent by the request, unsigned and ready for use.
17.	req.stale	It indicates whether the request is "stale," and is the opposite of req.fresh.
18.	req.subdomains	It represents an array of subdomains in the domain name of the request.
19.	req.xhr	A Boolean value that is true if the request's "x-requested-with" header field is "xmlhttprequest", indicating that the request was issued by a client library such as jQuery

# Request Object Methods

Following is a list of some generally used request object methods:

## req.accepts (types)

This method is used to check whether the specified content types are acceptable, based on the request's Accept HTTP header field.

### Examples:

1. req.accepts('html');
2. // => ?html?
3. req.accepts('text/html');
4. // => ?text/html?

## req.get(field)

This method returns the specified HTTP request header field.

### Examples:

1. req.get('Content-Type');
2. // => "text/plain"
3. req.get('content-type');
4. // => "text/plain"
5. req.get('Something');
6. // => undefined

## req.is(type)

This method returns true if the incoming request's "Content-Type" HTTP header field matches the MIME type specified by the type parameter.

### Examples:

1. // With Content-Type: text/html; charset=utf-8
2. req.is('html');
3. req.is('text/html');
4. req.is('text/\*');
5. // => true

## req.param(name [, defaultValue])

This method is used to fetch the value of param name when present.

**Examples:**

1. `// ?name=sasha`
2. `req.param('name')`
3. `// => "sasha"`
4. `// POST name=sasha`
5. `req.param('name')`
6. `// => "sasha"`
7. `// /user/sasha for /user/:name`
8. `req.param('name')`
9. `// => "sasha"`

# Express.js Response Object

The Response object (res) specifies the HTTP response which is sent by an Express app when it gets an HTTP request.

---

## What it does

- It sends response back to the client browser.
  - It facilitates you to put new cookies value and that will write to the client browser (under cross domain rule).
  - Once you `res.send()` or `res.redirect()` or `res.render()`, you cannot do it again, otherwise, there will be uncaught error.
- 

## Response Object Properties

Let's see some properties of response object.

Index	Properties	Description
1.	<code>res.app</code>	It holds a reference to the instance of the express application that is using the middleware.
2.	<code>res.headersSent</code>	It is a Boolean property that indicates if the app sent HTTP headers for the response.
3.	<code>res.locals</code>	It specifies an object that contains response local variables scoped to the request

## Response Object Methods

Following are some methods:

### Response Append method

**Syntax:**

1. `res.append(field [, value])`

This method appends the specified value to the HTTP response header field. That means if the specified value is not appropriate then this method redress that.

**Examples:**

1. `res.append('Link', ['<http://localhost/>', '<http://localhost:3000/>']);`
2. `res.append('Warning', '199 Miscellaneous warning');`

## Response Attachment method

**Syntax:**

1. `res.attachment([filename])`

This method facilitates you to send a file as an attachment in the HTTP response.

**Examples:**

1. `res.attachment('path/to/js_pic.png');`

## Response Cookie method

**Syntax:**

1. `res.cookie(name, value [, options])`

This method is used to set a cookie name to value. The value can be a string or object converted to JSON.

**Examples:**

1. `res.cookie('name', 'Aryan', { domain: '.xyz.com', path: '/admin', secure: true });`
2. `res.cookie('Section', { Names: [Aryan,Sushil,Priyanka] });`
3. `res.cookie('Cart', { items: [1,2,3] }, { maxAge: 900000 });`

## Response ClearCookie method

**Syntax:**

1. `res.clearCookie(name [, options])`

As the name specifies, the clearCookie method is used to clear the cookie specified by name.

### **Examples:**

#### **To set a cookie**

1. `res.cookie('name', 'Aryan', { path: '/admin' });`

#### **To clear a cookie:**

1. `res.clearCookie('name', { path: '/admin' });`

## Response Download method

### **Syntax:**

1. `res.download(path [, filename] [, fn])`

This method transfers the file at path as an "attachment" and enforces the browser to prompt user for download.

### **Example:**

1. `res.download('/report-12345.pdf');`

## Response End method

### **Syntax:**

1. `res.end([data] [, encoding])`

This method is used to end the response process.

### **Example:**

1. `res.end();`
2. `res.status(404).end();`

## Response Format method

### **Syntax:**

1. `res.format(object)`

This method performs content negotiation on the Accept HTTP header on the request object, when present.

### **Example:**

```
1. res.format({
2.   'text/plain': function(){
3.     res.send('hey');
4.   },
5.   'text/html': function(){
6.     res.send('
7. hey');
8.   },
9.   'application/json': function(){
10.    res.send({ message: 'hey' });
11.  },
12. 'default': function() {
13.   // log the request and respond with 406
14.   res.status(406).send('Not Acceptable');
15. }
16.});
```

## Response Get method

### Syntax:

```
1. res.get(field)
```

This method provides HTTP response header specified by field.

### Example:

```
1. res.get('Content-Type');
```

## Response JSON method:

### Syntax:

```
1. res.json([body])
```

This method returns the response in JSON format.

### Example:

```
1. res.json(null)
2. res.json({ name: 'ajeet' })
```

## Response JSONP method



**Syntax:**

1. `res.jsonp([body])`

This method returns response in JSON format with JSONP support.

**Examples:**

1. `res.jsonp(null)`
2. `res.jsonp({ name: 'ajeet' })`

## Response Links method

**Syntax:**

1. `res.links(links)`

This method populates the response's Link HTTP header field by joining the links provided as properties of the parameter.

**Examples:**

1. `res.links({`
2. `next: 'http://api.rnd.com/users?page=5',`
3. `last: 'http://api.rnd.com/users?page=10'`
4. `});`

## Response Location method

**Syntax:**

1. `res.location(path)`

This method is used to set the response location HTTP header field based on the specified path parameter.

**Examples:**

1. `res.location('http://xyz.com');`

## Response Redirect method

**Syntax:**

1. `res.redirect([status,] path)`

This method redirects to the URL derived from the specified path, with specified HTTP status

**Examples:**

1. `res.redirect('http://example.com');`

## Response Render method

**Syntax:**

1. `res.render(view [, locals] [, callback])`

This method renders a view and sends the rendered HTML string to the client.

**Examples:**

1. `// send the rendered view to the client`
2. `res.render('index');`
3. `// pass a local variable to the view`
4. `res.render('user', { name: 'aryan' }, function(err, html) {`
5.  `// ...`
6. `});`

## Response Send method

**Syntax:**

1. `res.send([body])`

This method is used to send HTTP response.

**Examples:**

1. `res.send(new Buffer('whoop'));`
2. `res.send({ some: 'json' });`
3. `res.send('`
4. `.....some html`
5. `');`

## Response sendFile method

**Syntax:**

1. `res.sendFile(path [, options] [, fn])`

This method is used to transfer the file at the given path. It sets the Content-Type response HTTP header field based on the filename's extension.

**Examples:**

1. `res.sendFile(fileName, options, function (err) {`
2.  `// ...`
3. `});`

## Response Set method

**Syntax:**

1. `res.set(field [, value])`

This method is used to set the response of HTTP header field to value.

**Examples:**

1. `res.set('Content-Type', 'text/plain');`
2.
3. `res.set({`
4.  `'Content-Type': 'text/plain',`
5.  `'Content-Length': '123',`
6. `});`

## Response Status method

**Syntax:**

1. `res.status(code)`

This method sets an HTTP status for the response.

**Examples:**

1. `res.status(403).end();`
2. `res.status(400).send('Bad Request');`

## Response Type method

**Syntax:**

## 1. res.type(type)

This method sets the content-type HTTP header to the MIME type.

### **Examples:**

1. `res.type('.html');` // => 'text/html'
2. `res.type('html');` // => 'text/html'
3. `res.type('json');` // => 'application/json'
4. `res.type('application/json');` // => 'application/json'
5. `res.type('png');` // => image/png:

# The Nodemailer Module

The Nodemailer module makes it easy to send emails from your computer.

The Nodemailer module can be downloaded and installed using npm:

```
C:\Users\Your Name>npm install nodemailer
```

After you have downloaded the Nodemailer module, you can include the module in any application:

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

## Send an Email

Now you are ready to send emails from your server.

Use the username and password from your selected email provider to send an email. This tutorial will show you how to use your Gmail account to send an email:

### Example

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

var transporter = nodemailer.createTransport({
  service: 'gmail',
  auth: {
    user: 'youremail@gmail.com',
    pass: 'yourpassword'
  }
});

var mailOptions = {
  from: 'youremail@gmail.com',
  to: 'myfriend@yahoo.com',
  subject: 'Sending Email using Node.js',
  text: 'That was easy!'
};
```

```
transporter.sendMail(mailOptions, function(error, info){
  if (error) {
    console.log(error);
  } else {
    console.log('Email sent: ' + info.response);
  }
});
```

And that's it! Now your server is able to send emails.

## Multiple Receivers

To send an email to more than one receiver, add them to the "to" property of the mailOptions object, separated by commas:

### Example

Send email to more than one address:

```
var mailOptions = {
  from: 'youremail@gmail.com',
  to: 'myfriend@yahoo.com, myotherfriend@yahoo.com',
  subject: 'Sending Email using Node.js',
  text: 'That was easy!'
}
```

## Send HTML

To send HTML formatted text in your email, use the "html" property instead of the "text" property:

### Example

Send email containing HTML:

```
var mailOptions = {
  from: 'youremail@gmail.com',
```

```
to: 'myfriend@yahoo.com',  
subject: 'Sending Email using Node.js',  
html: '<h1>Welcome</h1><p>That was easy!</p>'  
}
```

# Node.js Console

The Node.js console module provides a simple debugging console similar to JavaScript console mechanism provided by web browsers.

There are three console methods that are used to write any node.js stream:

1. `console.log()`
2. `console.error()`
3. `console.warn()`

```
console.log('Hello world');  
// format specifier  
console.log('Hello %s', ' world');  
console.error(new Error('Hell! This is a wrong method.'));  
const name = 'John';  
console.warn(`Don't mess with me ${name}! Don't mess with me!`);
```



# Node.js REPL

The term REPL stands for **Read Eval Print** and **Loop**. It specifies a computer environment like a window console or a Unix/Linux shell where you can enter the commands and the system responds with an output in an interactive mode.

## REPL Environment

The Node.js or node come bundled with REPL environment. Each part of the REPL environment has a specific work.

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

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

**Print:** It prints the result.

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

## How to start REPL

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

Multiple statments

```
var x = 0
undefined
> do {
... x++;
... console.log("x: " + x);
... } while ( x < 10 );
```

## Node.js Underscore Variable

You can also use underscore `_` to get the last result.

## Node.js REPL Commands

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

ctrl + c twice	It terminates the node repl.
ctrl + d	It terminates the node repl.
up/down keys	It is used to see command history and modify previous commands
tab keys	It specifies the list of current command.
.help	It specifies the list of all commands.
.break	It is used to exit from multi-line expressions.
.clear	It is used to exit from multi-line expressions.
.save filename	It saves current node repl session to a file.
.load filename	It is used to load file content in current node repl session.

## Node.js Exit REPL

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

# Node.js Package Manager

Node Package Manager provides two main functionalities:

- It provides online repositories for node.js packages/modules which are searchable on [search.npmjs.org](https://search.npmjs.org)
- It also provides command line utility to install Node.js packages, do version management and dependency management of Node.js packages.

The npm comes bundled with Node.js installables in versions after that v0.6.3. You can check the version by opening Node.js command prompt and typing the following command:

```
npm version
```

## Installing Modules using npm

Following is the syntax to install any Node.js module:

```
npm install <Module Name>
```

Let's install a famous Node.js web framework called express:

Open the Node.js command prompt and execute the following command:

```
npm install express
```

## Global vs Local Installation

By default, npm installs dependency in local mode. Here local mode specifies the folder where Node application is present. For example if you installed express module, it created node\_modules directory in the current directory where it installed express module.

You can use npm ls command to list down all the locally installed modules.

Open the Node.js command prompt and execute "npm ls":

Globally installed packages/dependencies are stored in system directory. Let's install express module using global installation. Although it will also produce the same result but modules will be installed globally.

Open Node.js command prompt and execute the following code:

```
npm install express -g
```

## Uninstalling a Module

To uninstall a Node.js module, use the following command:

```
npm uninstall express
```

## Searching a Module

"npm search express" command is used to search express or module.

```
npm search express
```

# Node.js Command Line Options

There is a wide variety of command line options in Node.js. These options provide multiple ways to run Node.js.

Let's see the list of Node.js command line options:

Index	Option	Description
1.	<code>v, --version</code>	It is used to print node's version.
2.	<code>-h, --help</code>	It is used to print node command line options.
3.	<code>-e, --eval "script"</code>	It evaluates the following argument as JavaScript. The modules which are predefined in the REPL can also be used in script.
4.	<code>-p, --print "script"</code>	It is identical to <code>-e</code> but prints the result.
5.	<code>-c, --check</code>	Syntax check the script without executing.
6.	<code>-i, --interactive</code>	It opens the REPL even if stdin does not appear to be a terminal.
7.	<code>-r, --require module</code>	It is used to preload the specified module at startup. It follows <code>require()</code> 's module resolution rules. Module may be either a path to a file, or a node module name.
8.	<code>--no-deprecation</code>	Silence deprecation warnings.
9.	<code>--trace-deprecation</code>	It is used to print stack traces for deprecations.
10.	<code>--throw-deprecation</code>	It throws errors for deprecations.
11.	<code>--no-warnings</code>	It silence all process warnings (including deprecations).
12.	<code>--trace-warnings</code>	It prints stack traces for process warnings (including deprecations).

13.	<code>--trace-sync-io</code>	It prints a stack trace whenever synchronous i/o is detected after the first turn of the event loop.
14.	<code>--zero-fill-buffers</code>	Automatically zero-fills all newly allocated buffer and slowbuffer instances.
15.	<code>--track-heap-objects</code>	It tracks heap object allocations for heap snapshots.
16.	<code>--prof-process</code>	It processes V8 profiler output generated using the v8 option <code>--prof</code> .
17.	<code>--V8-options</code>	It prints V8 command line options.
18.	<code>--tls-cipher-list=list</code>	It specifies an alternative default tls cipher list. (requires node.js to be built with crypto support. (default))
19.	<code>--enable-fips</code>	It enables fips-compliant crypto at startup. (requires node.js to be built with ./configure --openssl-fips)
20.	<code>--force-fips</code>	It forces fips-compliant crypto on startup. (cannot be disabled from script code.) (same requirements as <code>--enable-fips</code> )
21.	<code>--icu-data-dir=file</code>	It specifies ICU data load path. (Overrides <code>node_icu_data</code> )

## Node.js Command Line Options Examples

To see the version of the running Node:

Open Node.js command prompt and run command `node -v` or `node --version`

# Node.js Global Objects

Node.js global objects are global in nature and available in all modules. You don't need to include these objects in your application; rather they can be used directly. These objects are modules, functions, strings and object etc. Some of these objects aren't actually in the global scope but in the module scope.

A list of Node.js global objects are given below:

- `__dirname`
- `__filename`
- `Console`
- `Process`
- `Buffer`
- `setImmediate(callback[, arg][, ...])`
- `setInterval(callback, delay[, arg][, ...])`
- `setTimeout(callback, delay[, arg][, ...])`
- `clearImmediate(immediateObject)`
- `clearInterval(intervalObject)`
- `clearTimeout(timeoutObject)`

## Node.js `__dirname`

It is a string. It specifies the name of the directory that currently contains the code.

*File: global-example1.js*

1. `console.log(__dirname);`

Open Node.js command prompt and run the following code:

1. `node global-example1.js`

## Node.js `__filename`

It specifies the filename of the code being executed. This is the resolved absolute path of this code file. The value inside a module is the path to that module file.

*File: global-example2.js*

1. `console.log(__filename);`

Open Node.js command prompt and run the following code:

1. `node global-example2.js`



# Node.js OS

Node.js OS provides some basic operating-system related utility functions. Let's see the list generally used functions or methods.

Index	Method	Description
1.	<code>os.arch()</code>	This method is used to fetch the operating system CPU architecture.
2.	<code>os.cpus()</code>	This method is used to fetch an array of objects containing information about each cpu/core installed: model, speed (in MHz), and times (an object containing the number of milliseconds the cpu/core spent in: user, nice, sys, idle, and irq).
3.	<code>os.endianness()</code>	This method returns the endianness of the cpu. Its possible values are 'BE' for big endian or 'LE' for little endian.
4.	<code>os.freemem()</code>	This methods returns the amount of free system memory in bytes.
5.	<code>os.homedir()</code>	This method returns the home directory of the current user.
6.	<code>os.hostname()</code>	This method is used to returns the hostname of the operating system.
7.	<code>os.loadavg()</code>	This method returns an array containing the 1, 5, and 15 minute load averages. The load average is a time fraction taken by system activity, calculated by the operating system and expressed as a fractional number.
8.	<code>os.networkinterfaces()</code>	This method returns a list of network interfaces.
9.	<code>os.platform()</code>	This method returns the operating system platform of the running computer i.e. 'darwin', 'win32', 'freebsd',

		'linux', 'sunos' etc.
10.	os.release()	This method returns the operating system release.
11.	os.tmpdir()	This method returns the operating system's default directory for temporary files.
12.	os.totalmem()	This method returns the total amount of system memory in bytes.
13.	os.type()	This method returns the operating system name. For example 'linux' on linux, 'darwin' on os x and 'windows_nt' on windows.
14.	os.uptime()	This method returns the system uptime in seconds.
15.	os.userInfo([options])	This method returns a subset of the password file entry for the current effective user.

## Node.js OS Example 1

In this example, we are including some basic functions. Create a file named `os_example1.js` having the following code:

*File: os\_example1.js*

```

1. const os=require('os');
2. console.log("os.freemem(): \n",os.freemem());
3. console.log("os.homedir(): \n",os.homedir());
4. console.log("os.hostname(): \n",os.hostname());
5. console.log("os.endianness(): \n",os.endianness());
6. console.log("os.loadavg(): \n",os.loadavg());
7. console.log("os.platform(): \n",os.platform());
8. console.log("os.release(): \n",os.release());
9. console.log("os.tmpdir(): \n",os.tmpdir());
10. console.log("os.totalmem(): \n",os.totalmem());
11. console.log("os.type(): \n",os.type());
12. console.log("os.uptime(): \n",os.uptime());

```

Open Node.js command prompt and run the following code:

# Node.js Timer

Node.js Timer functions are global functions. You don't need to use `require()` function in order to use timer functions. Let's see the list of timer functions.

## Set timer functions:

- **`setImmediate()`**: It is used to execute `setImmediate`.
- **`setInterval()`**: It is used to define a time interval.
- **`setTimeout()`**: `()`- It is used to execute a one-time callback after delay milliseconds.

## Clear timer functions:

- **`clearImmediate(immediateObject)`**: It is used to stop an `immediateObject`, as created by `setImmediate`
- **`clearInterval(intervalObject)`**: It is used to stop an `intervalObject`, as created by `setInterval`
- **`clearTimeout(timeoutObject)`**: It prevents a `timeoutObject`, as created by `setTimeout`

## Node.js Timer `setInterval()` Example

This example will set a time interval of 1000 millisecond and the specified comment will be displayed after every 1000 millisecond until you terminate.

*File: timer1.js*

```
setInterval(function() {  
  console.log("setInterval: Hey! 1 millisecond completed!..");  
}, 1000);
```

Open Node.js command prompt and run the following code:

1. `node timer1.js`

*File: timer5.js*

```
var i =0;  
console.log(i);  
setInterval(function(){
```

```
i++;  
console.log(i);  
}, 1000);
```

Open Node.js command prompt and run the following code:

1. node timer5.js

## Node.js Timer setTimeout() Example

*File: timer1.js*

```
setTimeout(function() {  
  console.log("setTimeout: Hey! 1000 millisecond completed!..");  
}, 1000);
```

Open Node.js command prompt and run the following code:

1. node timer1.js

This example shows time out after every 1000 millisecond without setting a time interval. This example uses the recursion property of a function.

*File: timer2.js*

```
var recursive = function () {  
  console.log("Hey! 1000 millisecond completed!..");  
  setTimeout(recursive,1000);  
}  
recursive();
```

Open Node.js command prompt and run the following code:

1. node timer2.js

## Node.js setInterval(), setTimeout() and clearTimeout()

Let's see an example to use clearTimeout() function.

*File: timer3.js*

```
function welcome () {  
  console.log("Welcome to JavaTpoint!");  
}  
var id1 = setTimeout(welcome,1000);  
var id2 = setInterval(welcome,1000);  
clearTimeout(id1);  
//clearInterval(id2);
```

Open Node.js command prompt and run the following code:

1. node timer3.js

You can see that the above example is recursive in nature. It will terminate after one step if you use ClearInterval.

## Node.js setInterval(), setTimeout() and clearInterval()

Let's see an example to use clearInterval() function.

*File: timer3.js*

```
function welcome () {  
  console.log("Welcome to JavaTpoint!");  
}  
var id1 = setTimeout(welcome,1000);  
var id2 = setInterval(welcome,1000);  
//clearTimeout(id1);  
clearInterval(id2);
```

Open Node.js command prompt and run the following code:

1. node timer3.js

# Node.js Net

Node.js provides the ability to perform socket programming. We can create chat application or communicate client and server applications using socket programming in Node.js. The Node.js net module contains functions for creating both servers and clients.

## Node.js Net Example

In this example, we are using two command prompts:

- Node.js command prompt for server.
- Window's default command prompt for client.

### server:

*File: net\_server.js*

```
const net = require('net');
var server = net.createServer((socket) => {
  socket.end('goodbye\n');
}).on('error', (err) => {
  // handle errors here
  throw err;
});
// grab a random port.
server.listen(() => {
  address = server.address();
  console.log('opened server on %j', address);
});
```

Open Node.js command prompt and run the following code:

1. node net\_server.js

### client:

*File: net\_client.js*

```
const net = require('net');
const client = net.connect({port: 50302}, () => { //use same port of server
```

```
    console.log('connected to server!');
    client.write('world!\r\n');
  });
  client.on('data', (data) => {
    console.log(data.toString());
    client.end();
  });
  client.on('end', () => {
    console.log('disconnected from server');
  });
```

Open Node.js command prompt and run the following code:

1. node net\_client.js