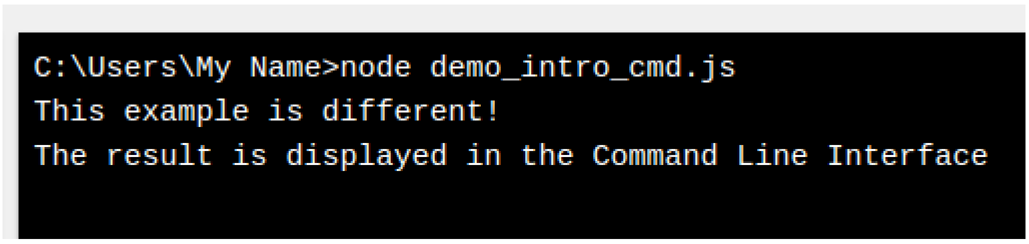
demo_intro_cmd.js:

```
console.log('This example is different!');
console.log('The result is displayed in the Command Line
Interface');
```

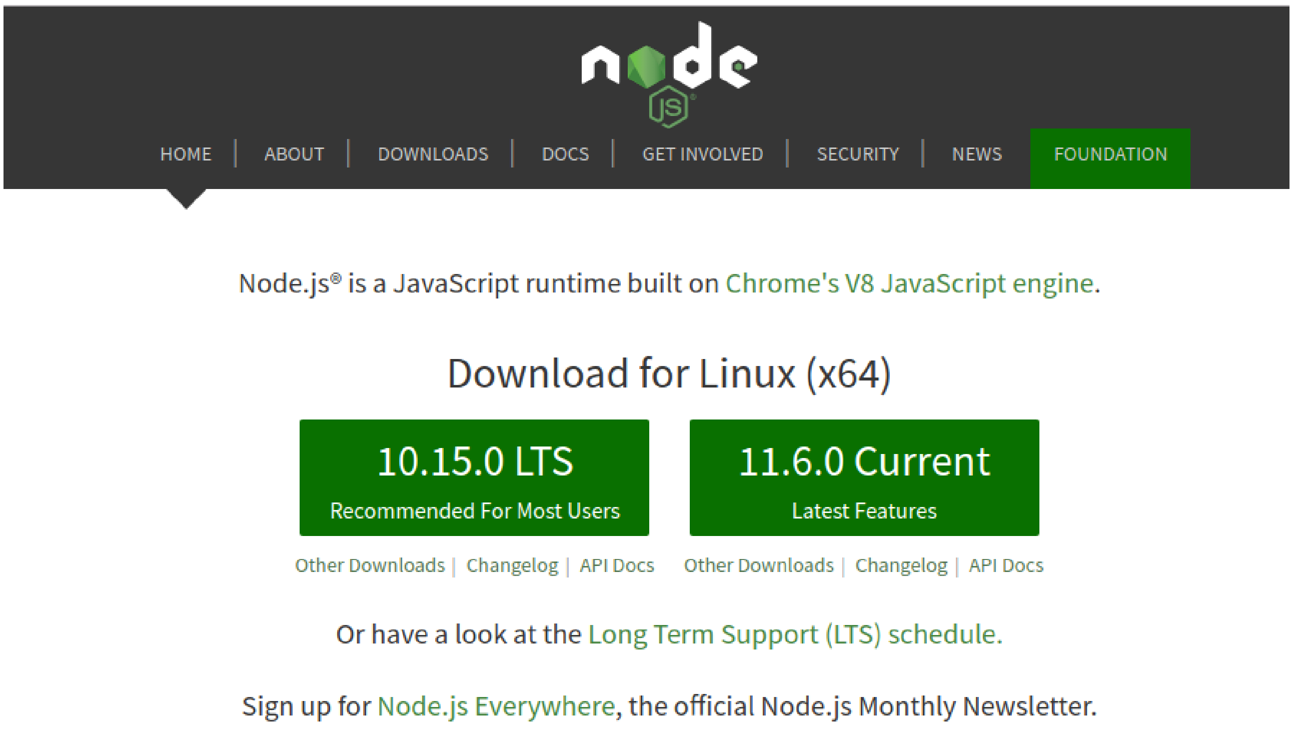


```
C:\Users\My Name>node demo_intro_cmd.js
This example is different!
The result is displayed in the Command Line Interface
```

Download Node.js

Download Node.js from the official Node.js web site: https://nodejs.org

https://nodejs.org/en/



Node.js® is a JavaScript runtime built on Chrome's V8 JavaScript engine.

Download for Linux (x64)

10.15.0 LTS
Recommended For Most Users

11.6.0 Current
Latest Features

Other Downloads | Changelog | API Docs

Other Downloads | Changelog | API Docs

Or have a look at the Long Term Support (LTS) schedule.

Sign up for Node.js Everywhere, the official Node.js Monthly Newsletter.

What is Node.js?

- Node.js is an open source server environment
- Node.js is free
- Node.js runs on various platforms (Windows, Linux, Unix, Mac OS X, etc.)
- Node.js uses JavaScript on the server

Why Node.js?

> **Node.js uses asynchronous programming!**

Why Node.js?

> **Node.js uses asynchronous programming!**

A common task for a web server can be to open a file on the server and return the content to the client.

Here is how PHP or ASP handles a file request:

1. Sends the task to the computer's file system.
2. Waits while the file system opens and reads the file.
3. Returns the content to the client.
4. Ready to handle the next request.

Here is how Node.js handles a file request:

1. Sends the task to the computer's file system.
2. Ready to handle the next request.
3. When the file system has opened and read the file, the server returns the content to the client.

Node.js eliminates the waiting, and simply continues with the next request.

Node.js runs single-threaded, non-blocking, asynchronously programming, which is very memory efficient.

What Can Node.js Do?

- Node.js can generate dynamic page content
- Node.js can create, open, read, write, delete, and close files on the server
- Node.js can collect form data
- Node.js can add, delete, modify data in your database

What is a Node.js File?

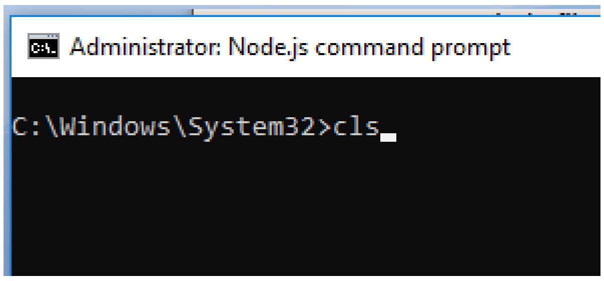
- Node.js files contain tasks that will be executed on certain events
- A typical event is someone trying to access a port on the server
- Node.js files must be initiated on the server before having any effect
- Node.js files have extension ".js"

....

Node js version

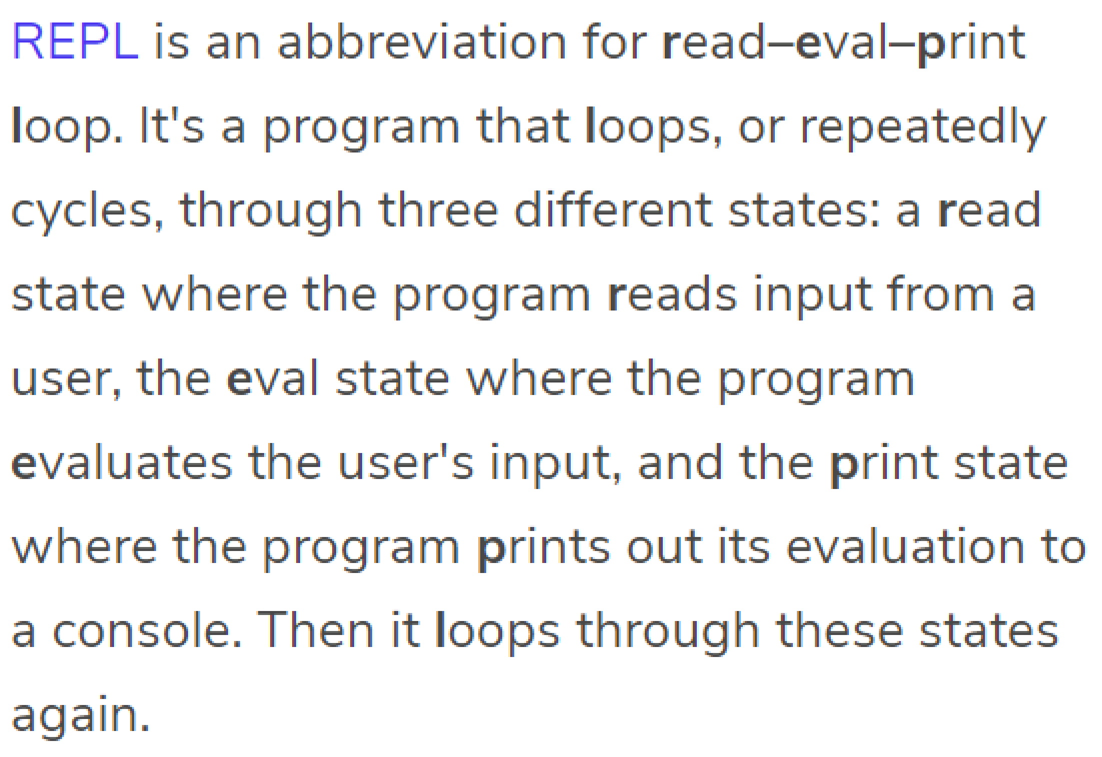
Let's see what version of Node we have installed. Type `node -v` in the terminal and then press and hit enter (or return).

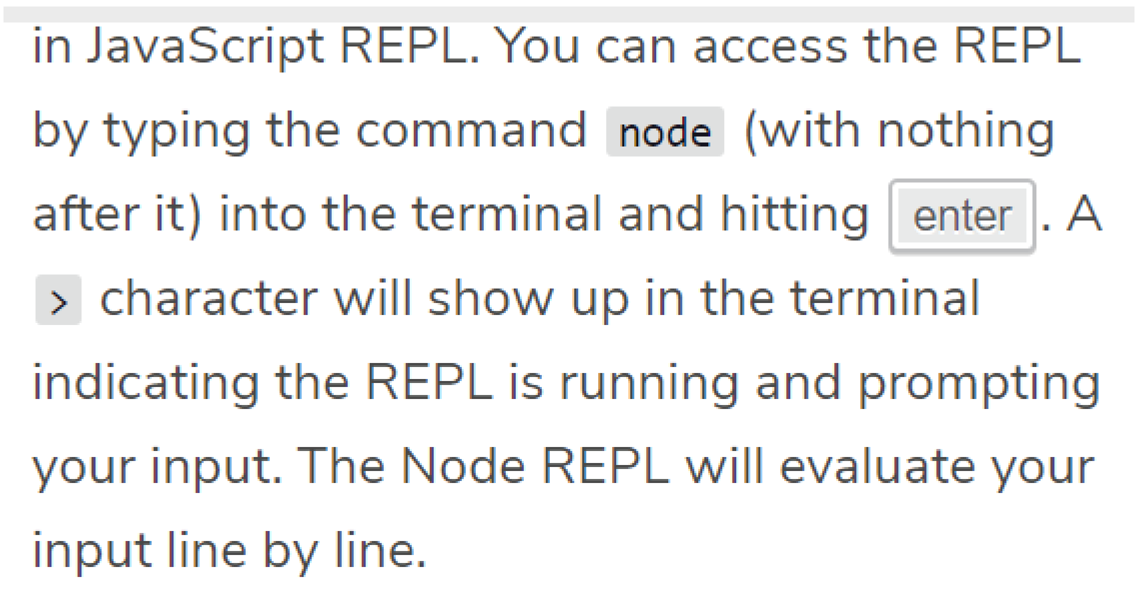
cls to clear terminal via windows


Administrator: Node.js command prompt

```
C:\Windows\System32>cls
```

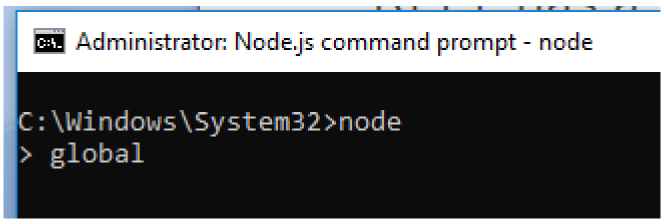
The Node REPL


REPL is an abbreviation for **r**ead–**e**val–**p**rint **l**oop. It's a program that **l**oops, or repeatedly cycles, through three different states: a **r**ead state where the program **r**eads input from a user, the **e**val state where the program **e**valuates the user's input, and the **p**rint state where the program **p**rints out its evaluation to a console. Then it **l**oops through these states again.

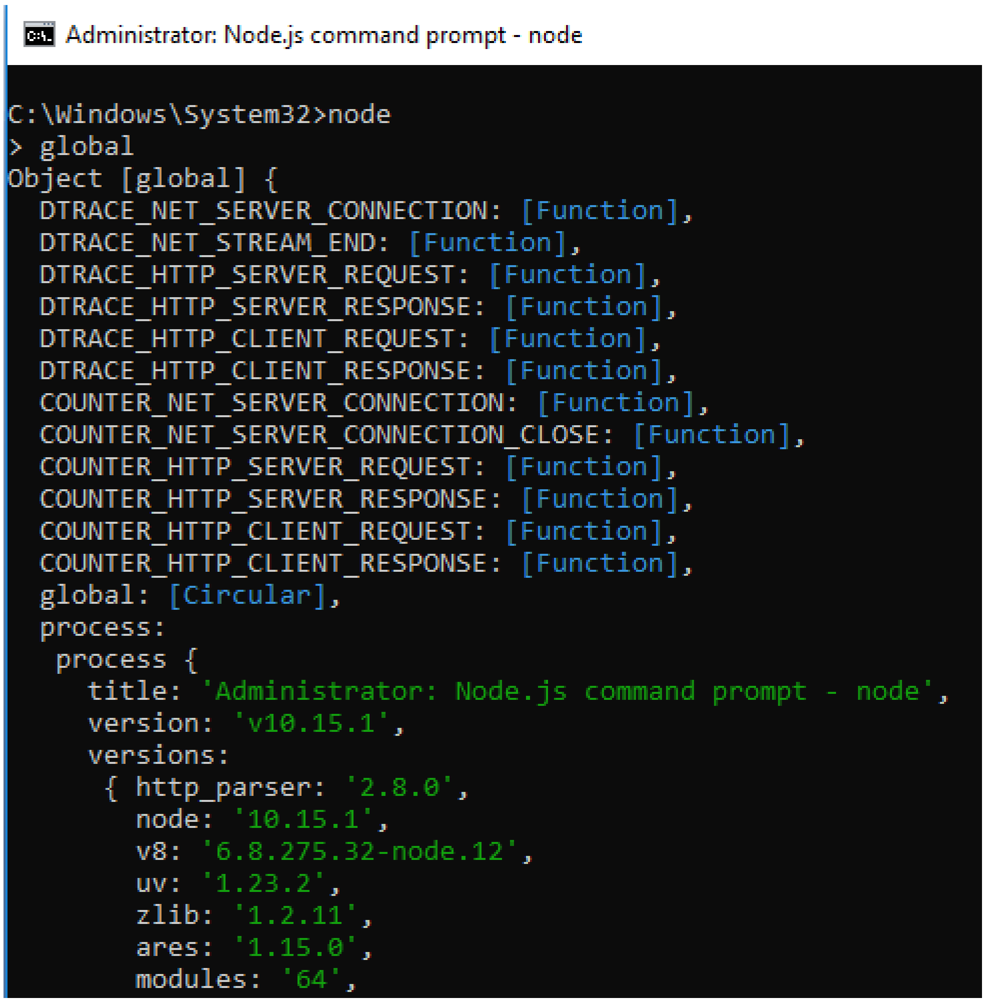

in JavaScript REPL. You can access the REPL by typing the command `node` (with nothing after it) into the terminal and hitting enter. A `>` character will show up in the terminal indicating the REPL is running and prompting your input. The Node REPL will evaluate your input line by line.

By default, you indicate the input is ready for eval when you hit enter. If you'd like to type multiple lines and then have them evaluated at once you can type `.editor` while in the REPL. Once in "editor" mode, you can type CONTROL D when you're ready for the input to be evaluated. Each session of the REPL has a single shared memory; you can access any variables or functions you define until you exit the REPL.

The Node environment contains a number of Node-specific global elements in addition to those built into the JavaScript language. Every Node-specific global property sits inside the the Node `global` object. This object contains a number of useful properties and methods that are available anywhere in the Node environment.

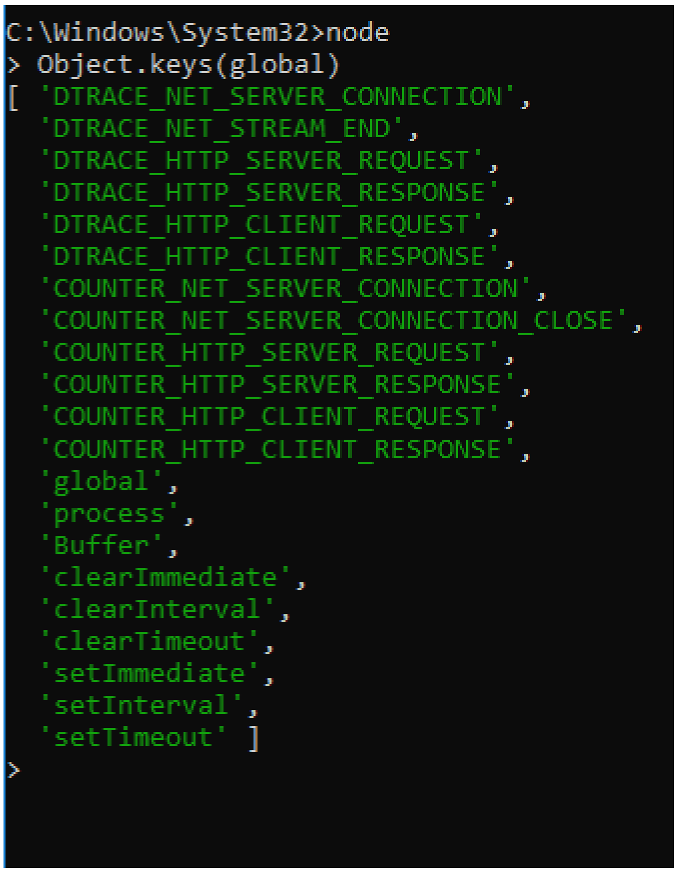


```
C:\Windows\System32>node
> global
```


Administrator: Node.js command prompt - node

```
C:\Windows\System32>node
> global
Object [global] {
  DTRACE_NET_SERVER_CONNECTION: [Function],
  DTRACE_NET_STREAM_END: [Function],
  DTRACE_HTTP_SERVER_REQUEST: [Function],
  DTRACE_HTTP_SERVER_RESPONSE: [Function],
  DTRACE_HTTP_CLIENT_REQUEST: [Function],
  DTRACE_HTTP_CLIENT_RESPONSE: [Function],
  COUNTER_NET_SERVER_CONNECTION: [Function],
  COUNTER_NET_SERVER_CONNECTION_CLOSE: [Function],
  COUNTER_HTTP_SERVER_REQUEST: [Function],
  COUNTER_HTTP_SERVER_RESPONSE: [Function],
  COUNTER_HTTP_CLIENT_REQUEST: [Function],
  COUNTER_HTTP_CLIENT_RESPONSE: [Function],
  global: [Circular],
  process:
   process {
     title: 'Administrator: Node.js command prompt - node',
     version: 'v10.15.1',
     versions:
      { http_parser: '2.8.0',
        node: '10.15.1',
        v8: '6.8.275.32-node.12',
        uv: '1.23.2',
        zlib: '1.2.11',
        ares: '1.15.0',
        modules: '64',
```

- Woah... it looks huge. A lot of that is because of the `global.process` object. Check out an easier to read list of the properties on the `global` object with `Object.keys(global)`.



```
C:\Windows\System32>node
> Object.keys(global)
[ 'DTRACE_NET_SERVER_CONNECTION',
  'DTRACE_NET_STREAM_END',
  'DTRACE_HTTP_SERVER_REQUEST',
  'DTRACE_HTTP_SERVER_RESPONSE',
  'DTRACE_HTTP_CLIENT_REQUEST',
  'DTRACE_HTTP_CLIENT_RESPONSE',
  'COUNTER_NET_SERVER_CONNECTION',
  'COUNTER_NET_SERVER_CONNECTION_CLOSE',
  'COUNTER_HTTP_SERVER_REQUEST',
  'COUNTER_HTTP_SERVER_RESPONSE',
  'COUNTER_HTTP_CLIENT_REQUEST',
  'COUNTER_HTTP_CLIENT_RESPONSE',
  'global',
  'process',
  'Buffer',
  'clearImmediate',
  'clearInterval',
  'clearTimeout',
  'setImmediate',
  'setInterval',
  'setTimeout' ]
>
```

You'll learn more about the `global` object as you explore Node, but remember that, at its core, it's just a JavaScript object!

Core Modules and Local Modules

Essentially, a module is a collection of code located in a file. Instead of having an entire program located in a single file, code is organized into separate files and combined through *requiring* them where needed using the `require()` function.

To save developers from having to reinvent the wheel each time, Node has several modules included within the environment to efficiently perform common tasks. These are known as the *core modules*. The core modules are defined within Node.js's source and are located in the lib/ folder. Core modules are required by passing a string with the name of the module into the `require()` function:

```
// Require in the 'events' core module:
let events = require('events');
```

the `require()`function includes some interesting logic "under the hood." The `require()` function will first check to if its argument is a core module, if not, it will move on to different attempts to locate it.

Let's walk through the process of requiring a local module:

```
// dog.js
module.exports = class Dog {

  constructor(name) {
    this.name = name;
  }

  praise() {
    return `Good dog, ${this.name}!`;
  }
};
```

Above, in the **dog.js** file, we assign the `Dog` class as the value of `module.exports`. Each JavaScript file in the Node environment has a special JavaScript object called `module.exports`. It holds everything in that file, or module, that's available to be required into a different file.

```
// app.js
let Dog = require('./dog.js');
const tadpole = new Dog('Tadpole');
console.log(tadpole.praise());
```

Note:

in the path to the module. The `require()` function has some other quirks, like assuming file extensions if none are provided; this means we could have written `let Dog = require('./dog');` in place of `let Dog = require('./dog.js');` in the code above, and the `require()` function would have still correctly located and required in **dog.js**.

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.

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:

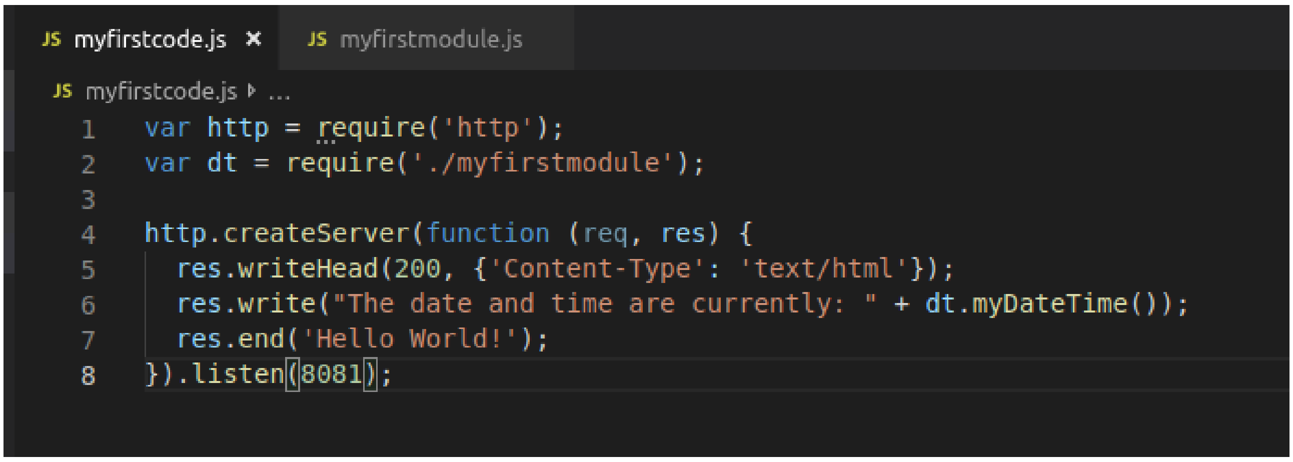
Example

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.

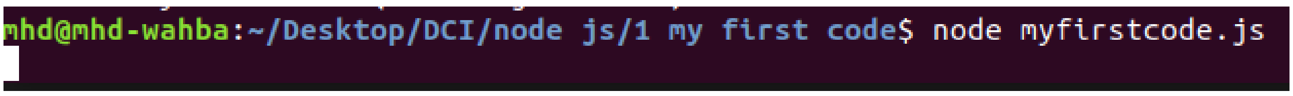
Save the code above in a file called "myfirstmodule.js"



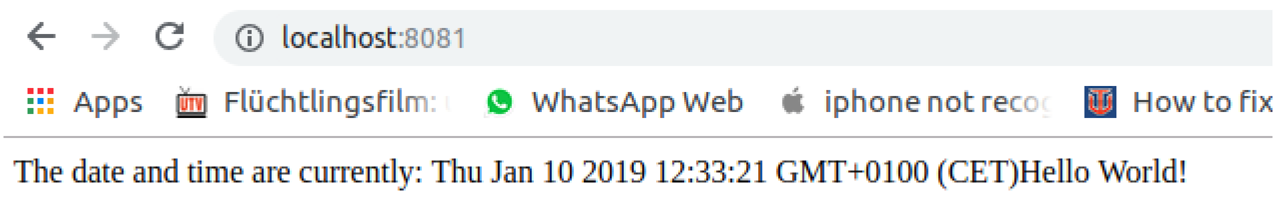
```
var http = require('http');
var dt = require('./myfirstmodule');

http.createServer(function (req, res) {
  res.writeHead(200, {'Content-Type': 'text/html'});
  res.write("The date and time are currently: " + dt.myDateTime());
  res.end('Hello World!');
}).listen(8081);
```

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

http://localhost:8080



```
mhd@mhd-wahba:~/Desktop/DCI/node js/1 my first code$ node myfirstcode.js
```



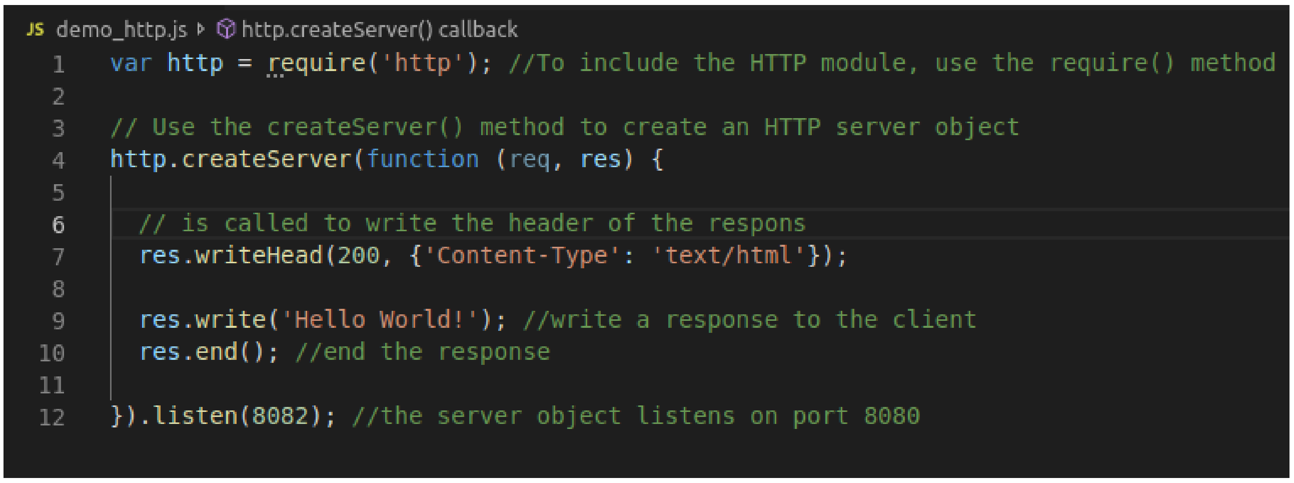
The date and time are currently: Thu Jan 10 2019 12:33:21 GMT+0100 (CET)Hello World!

The Built-in HTTP Module

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:



```javascript
var http = require('http'); //To include the HTTP module, use the require() method

// Use the createServer() method to create an HTTP server object
http.createServer(function (req, res) {

  // is called to write the header of the respons
  res.writeHead(200, {'Content-Type': 'text/html'});

  res.write('Hello World!'); //write a response to the client
  res.end(); //end the response

}).listen(8082); //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.

Add an HTTP Header

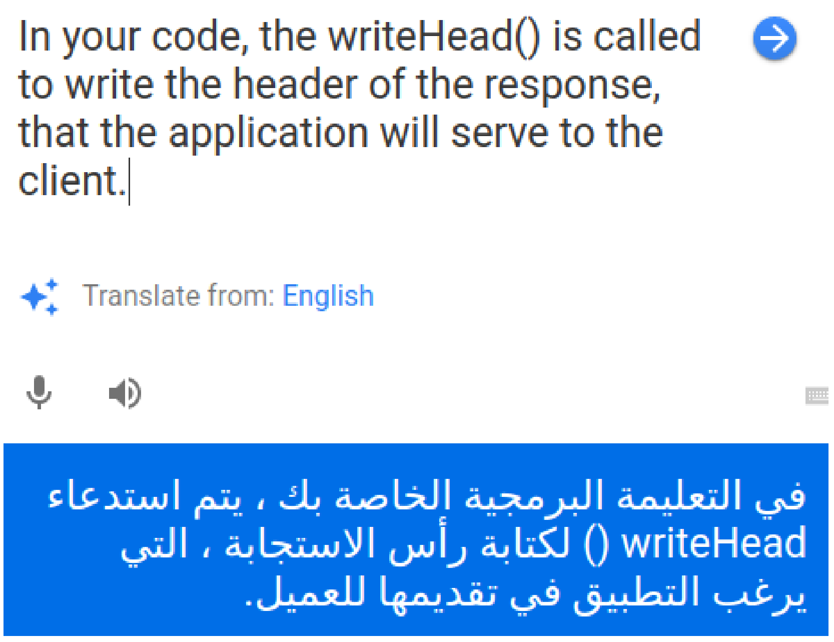
If the response from the HTTP server is supposed to be displayed as HTML, you should include an HTTP header with the correct content type:

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);
```

Run example »

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.


In your code, the writeHead() is called to write the header of the response, that the application will serve to the client.

في التعليمة البرمجية الخاصة بك ، يتم استدعاء writeHead () لكتابة رأس الاستجابة ، التي يرغب التطبيق في تقديمها للعميل.

The callback function begins by calling the `response.writeHead()` method. This method sends an HTTP status code and a collection of response headers back to the client. The status code is used to indicate the result of the request. For example, everyone has encountered a 404 error before, indicating that a page could not be found. The example server returns the code 200, which indicates success.

Along with the status code, the server returns a number of HTTP headers which define the parameters of the response. If you do not specify headers, Node.js will implicitly send them for you. The example server specifies only the `Content-Type`

10.2.1 200 OK

The request has succeeded.

10.4.5 404 Not Found

The server has not found anything matching the Request-URI.

The `content-type:text/html` **is** required for browsers to recognise a page as HTML. The `content-type` is not part of the HTML5 (or HTML4) spec; it's part of the HTTP headers; it is what tells the browser what kind of content is is receiving.

If you provide the wrong content type, the browser is likely to be unable to display your page, or may simply display the content it receives as raw text. The content type is a mandatory part of the HTTP protocol.

Read the Query String

query=question=inquiry=request

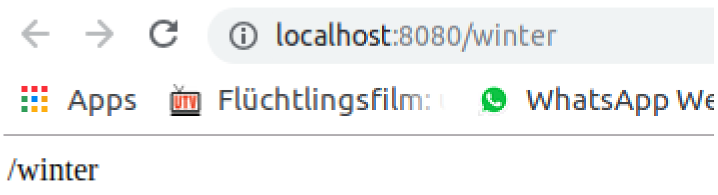
```
req.url)
```

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

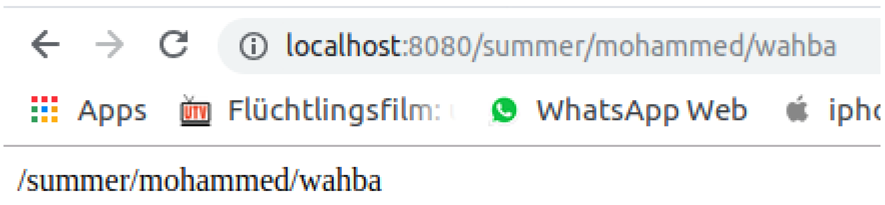
This object has a property called "url" which holds the part of the url that comes after the domain name:

ex:

```
//print of This object has a property called "url" which holds the part of the url that comes after the domain name
res.write(req.url);
```



```
/winter
```

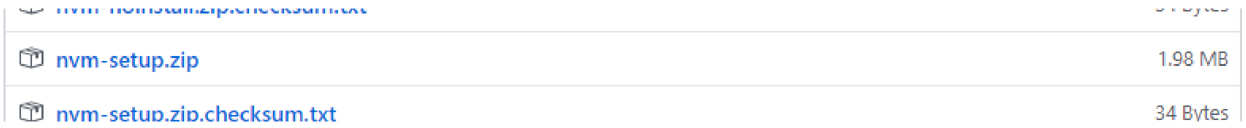


```
/summer/mohammed/wahba
```

To update node.js

https://github.com/coreybutler/nvm-windows/releases

and install this: nvm-setup.zip

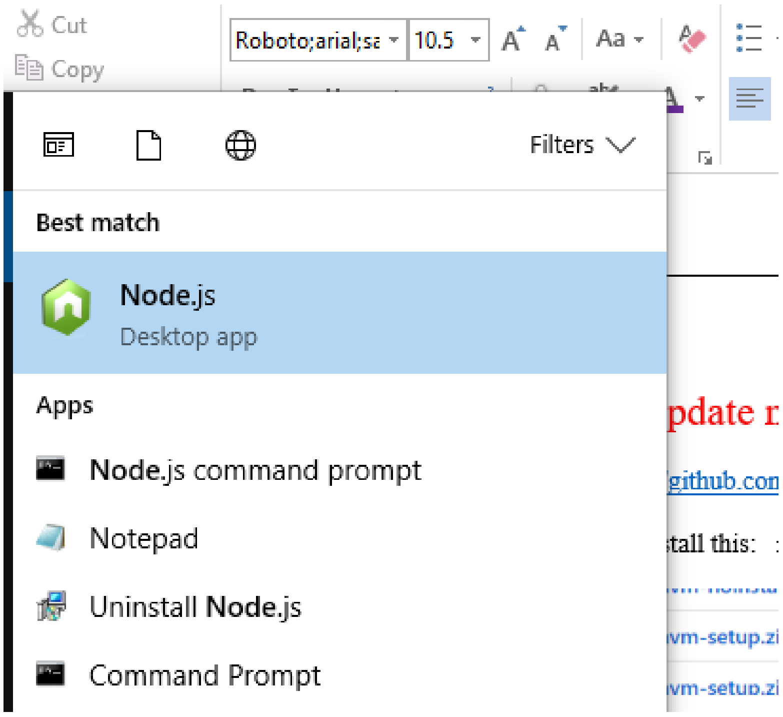


nvm-setup.zip	1.98 MB
nvm-setup.zip.checksum.txt	34 Bytes

REPL   Read Evaluate Print Loop

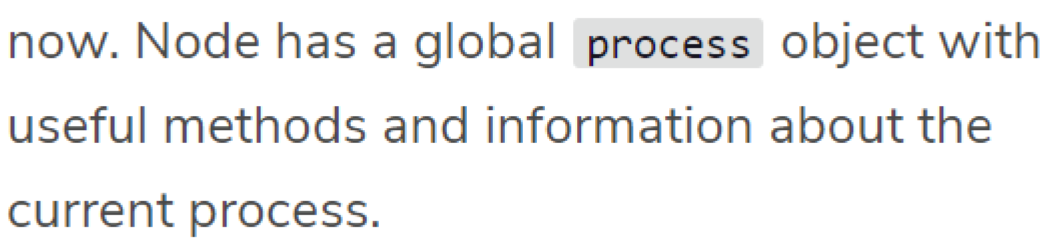
To execute the code of js (node js).

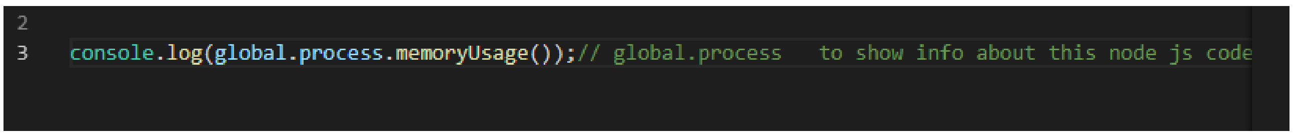
To test the code, like console log in Browser .

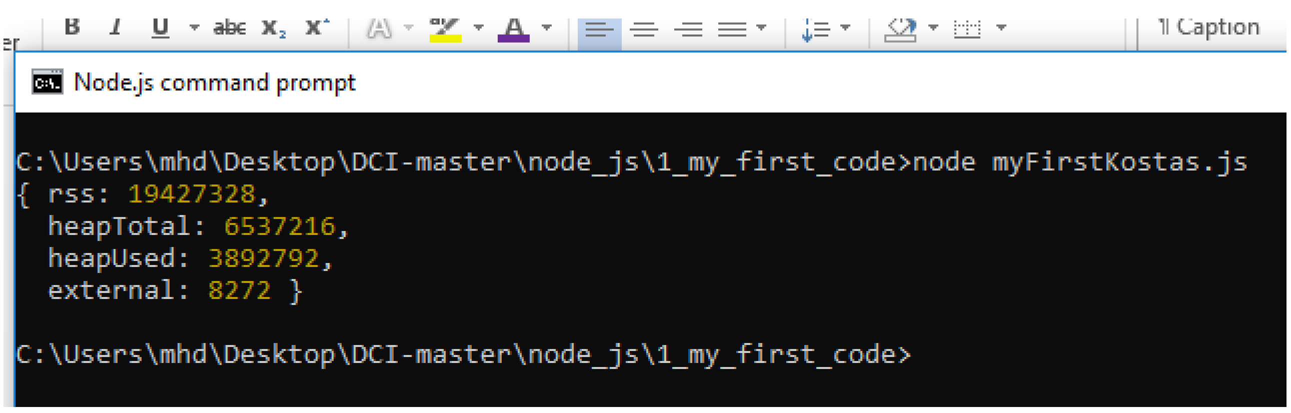


---

global.process


now. Node has a global `process` object with useful methods and information about the current process.


```
console.log(global.process.memoryUsage());// global.process   to show info about this node js code
```



```
C:\Users\mhd\Desktop\DCI-master\node_js\1_my_first_code>node myFirstKostas.js
{ rss: 19427328,
  heapTotal: 6537216,
  heapUsed: 3892792,
  external: 8272 }

C:\Users\mhd\Desktop\DCI-master\node_js\1_my_first_code>
```

In computer science, a *process* is the instance of a computer program that is being executed. You can open Task Manager if you're on a Windows machine or Activity Monitor from a Mac to see information about the various processes running on your computer right now. Node has a global `process` object with useful methods and information about the current process.

The `process.env` property is an object which stores and controls information about the environment in which the process is currently running. For example, the `process.env` object contains a `PWD` property which holds a string with the directory in which the current process is located. It can be useful to have some `if/else` logic in a program depending on the current environment— a web application in a development phase might perform different tasks than when it's live to users. We could store this information on the `process.env`. One convention is to add a property to `process.env` with the key `NODE_ENV` and a value of either `production` or `development`.

```
if (process.env.NODE_ENV ===
'development'){
  console.log('Testing! Testing! Does
everything work?');
}
```

The `process.memoryUsage()` returns information on the CPU demands of the current process. It returns a property that looks similar to this:

```
{ rss: 26247168,
  heapTotal: 5767168,
  heapUsed: 3573032,
  external: 8772 }
```

Heap can mean different things in different contexts: a heap can refer to a specific data structure, but it can also refer to the a block of computer memory. `process.memoryUsage().heapUsed` will return a number representing how many bytes of memory the current process is using.

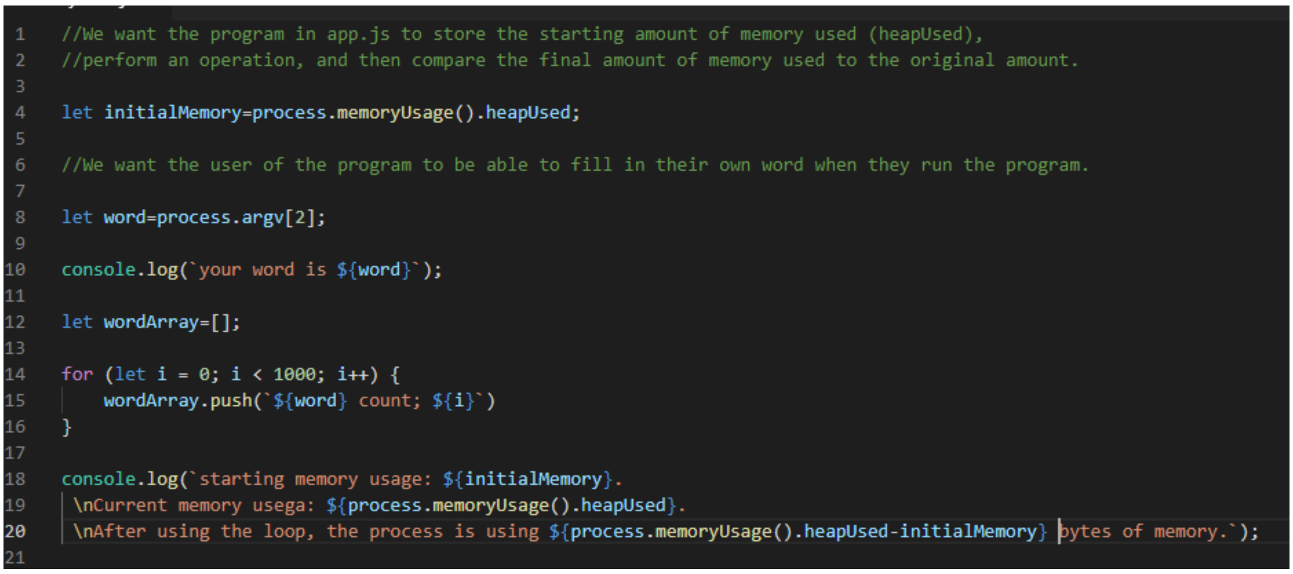
The `process.argv` property holds an array of command line values provided when the current process was initiated. The first element in the array is the absolute path to Node, which ran the process. The second element in the array is the path to the file that's running. The following elements will be any command line arguments provided when the process was initiated. Command line arguments are separated from one another with spaces.

```
node myProgram.js testing several features
```

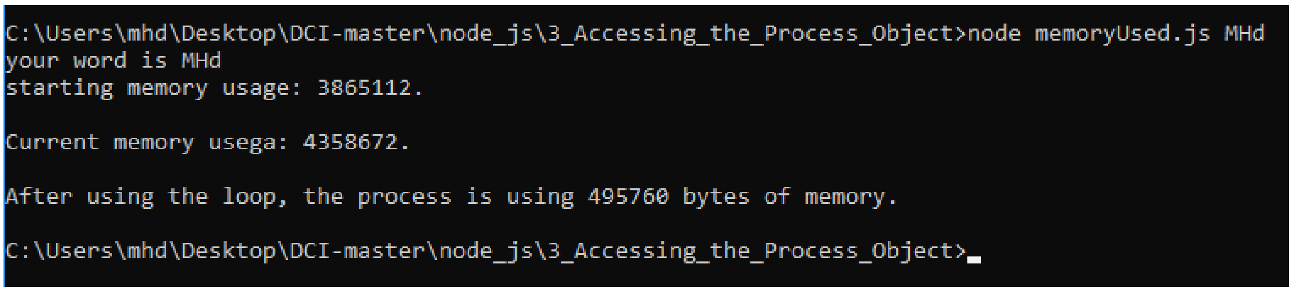
```
console.log(process.argv[3]); // Prints
'several'
```

We've only covered a few of the properties of the `process` object, so make sure to check out the documentation on the `process` object to learn more about it and explore some of its other methods and properties.

Ex:



```
//We want the program in app.js to store the starting amount of memory used (heapUsed),
//perform an operation, and then compare the final amount of memory used to the original amount.

let initialMemory=process.memoryUsage().heapUsed;

//We want the user of the program to be able to fill in their own word when they run the program.

let word=process.argv[2];

console.log(`your word is ${word}`);

let wordArray=[];

for (let i = 0; i < 1000; i++) {
    wordArray.push(`${word} count; ${i}`)
}

console.log(`starting memory usage: ${initialMemory}.
 \nCurrent memory usega: ${process.memoryUsage().heapUsed}.
 \nAfter using the loop, the process is using ${process.memoryUsage().heapUsed-initialMemory} bytes of memory.`);

```



```
C:\Users\mhd\Desktop\DCI-master\node_js\3_Accessing_the_Process_Object>node memoryUsed.js MHd
your word is MHd
starting memory usage: 3865112.

Current memory usega: 4358672.

After using the loop, the process is using 495760 bytes of memory.

C:\Users\mhd\Desktop\DCI-master\node_js\3_Accessing_the_Process_Object>
```

---

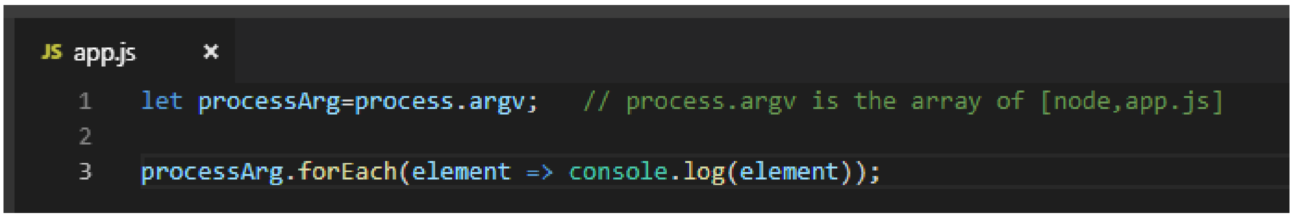
process.exit();

to stop to execute the code.

---

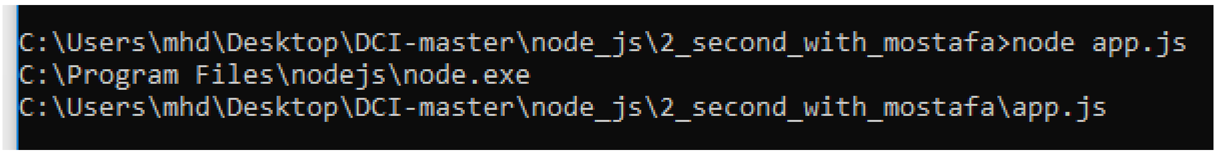
process.argv

Process.argv is the array of [node, app.js]



app.js

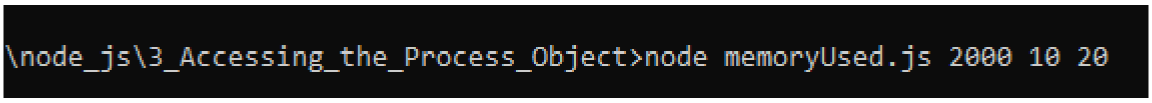
```
let processArg=process.argv;   // process.argv is the array of [node,app.js]

processArg.forEach(element => console.log(element));
```



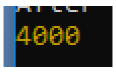
```
C:\Users\mhd\Desktop\DCI-master\node_js\2_second_with_mostafa>node app.js
C:\Program Files\nodejs\node.exe
C:\Users\mhd\Desktop\DCI-master\node_js\2_second_with_mostafa\app.js
```

Or from Kostas: slice + spread function
3_Accessing_the_Process_Object

```
// another way from kostas
const myArgArray=process.argv.slice(2);  // it will take copy from process.argv
array, start from index 2. to new array myArgArray
const calculate=(salary, numOfMonth,
percentDeducted)=>(percentDeducted/100)*salary*numOfMonth;
console.log(calculate(...myArgArray));
```



```
\node_js\3_Accessing_the_Process_Object>node memoryUsed.js 2000 10 20
```



```
4000
```

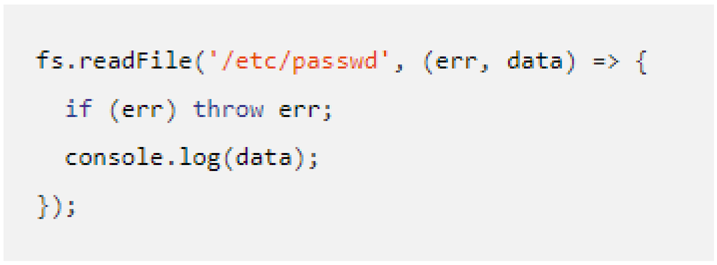
filesystem core module

to read and write files

const fs=require('fs'); // to work with filesystem core module

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

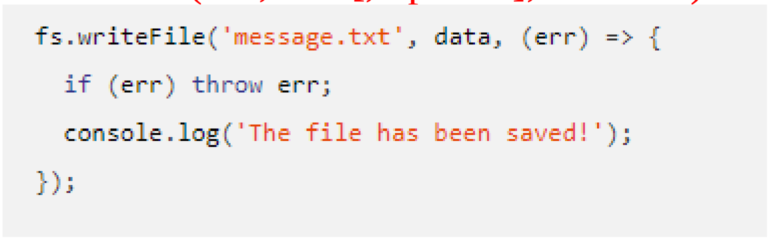
Asynchronously reads the entire contents of a file.



```
fs.readFile('/etc/passwd', (err, data) => {
  if (err) throw err;
  console.log(data);
});
```

```
fs.readFile('./ex.txt',(err,data)=>{

    if (err) console.log(err);
    else console.log(data.toString('utf8'));

});
```

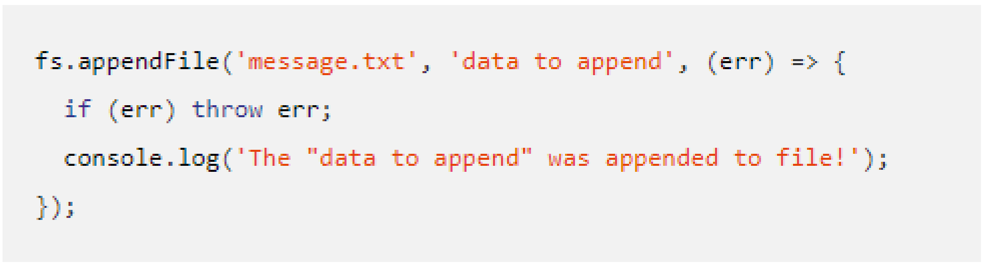
fs.writeFile(file, data[, options], callback)



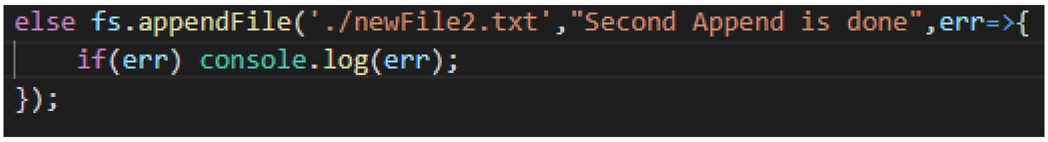
```
fs.writeFile('message.txt', data, (err) => {
  if (err) throw err;
  console.log('The file has been saved!');
});
```

```
fs.writeFile('./newFile.txt',`Hallo Mohammed`,err=>{

    if (err) console.log(err);
    else console.log('Writing done');

});
```

fs.appendFile(path, data[, options], callback)



```
fs.appendFile('message.txt', 'data to append', (err) => {
  if (err) throw err;
  console.log('The "data to append" was appended to file!');
});
```



```
else fs.appendFile('./newFile2.txt',"Second Append is done",err=>{
    if(err) console.log(err);
});
```

Promise core module

1 create promise module

```
const util=require('util'); // to work with promise core module
```

2 const promiseVarible = util.promisify(function that returns callback)

```
const readPromiseWay=util.promisify(fs.readFile);
```

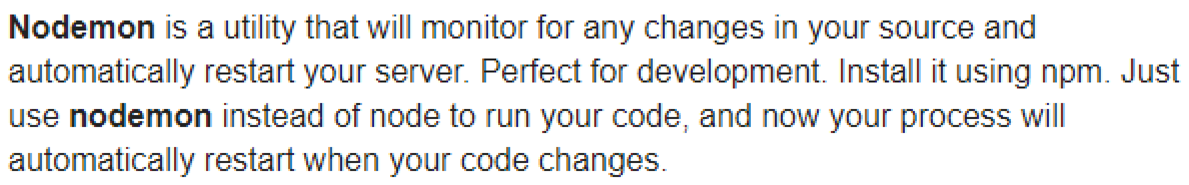
Whole code:

```
const fs=require('fs'); // to work with filesystem core module
const util=require('util'); // to work with promise core module


const readPromiseWay = util.promisify(fs.readFile);
const writePromiseWay = util.promisify(fs.writeFile);
const appendPromiseWay = util.promisify(fs.appendFile);

const dealWithFiles = async (file1, file2, file3) => {

  try {
    const data = await readPromiseWay(file1);
    console.log('Read done!');
    await writePromiseWay('./empty.txt', data.toString('utf8'));
    console.log('Write done!');
    await appendPromiseWay('./empty.txt', 'That was added by Jake, minutes before
the break');
    console.log('Append operation finished!');
    const results = await readPromiseWay('./lorem.txt');
    console.log('Read lorem ipsum');
    await appendPromiseWay('./empty.txt', ' ' + results.toString('utf8'));
    console.log('Appended the lorem ipsum');
} catch(error) {
    console.log(error);
  }
}

dealWithFiles('./example.txt', './empty.txt', './lorem.txt');

console.log('Execute first!');
```

3 promiseVarible(path)
    .then(one argument arrow function + return asyn function)
    .then(one argument arrow function + return asyn function)
    .then(one argument arrow function + return asyn function)
    .then(one argument arrow function)
    .catch(err => console.log(err))

*When first then done,so the second then will take the previous return and executes.
* any then has error, so it will go next thens and go catch.
* no then inside then
* any error will happen in all thens, it will pass to catch(err=>)

Nodemon
npm install -g nodemon


Nodemon is a utility that will monitor for any changes in your source and automatically restart your server. Perfect for development. Install it using npm. Just use **nodemon** instead of node to run your code, and now your process will automatically restart when your code changes.

Postman
https://www.getpostman.com/downloads/

What is Postman ?

- It is an application for testing the APIs.
- It acts as a client while testing the application developed in RESTful state.
- You can create an environment, write test cases, share APIs,etc.

When it is Used ?

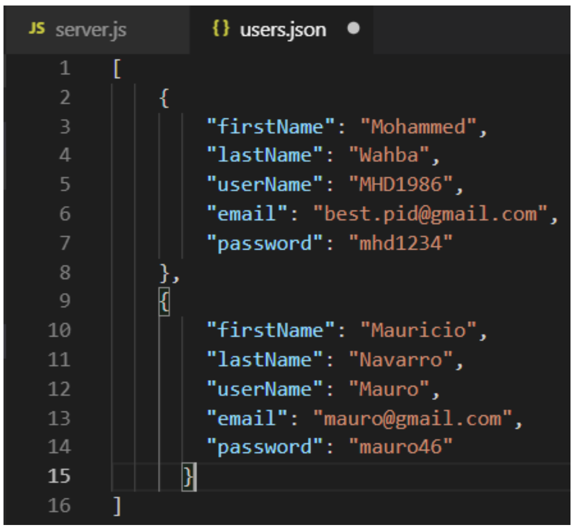
- Whenever you wish to test your applications behaviour for a particular API endpoint, after being requested by client.
- To view a response the server returns after requesting the API endpoint.
- Customizing the requests made to server, and test the server's response in different circumstances.

Who uses it ?

- Developer - While building any application, you need to test the endpoints. What should be the response and the format in which client-end gets the response after being requested.
- Tester - They can set the various environments in Postman and test the applications behaviour under certain circumstances.

Restful API

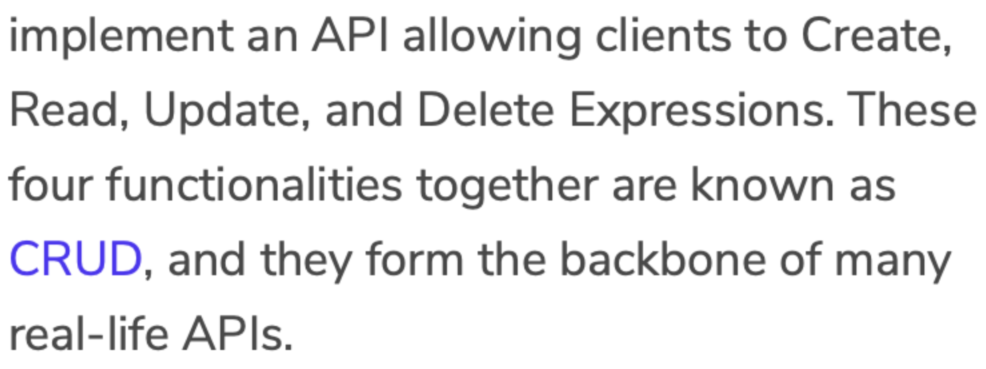
To get, update, and delete data.



```json
[
    {
        "firstName": "Mohammed",
        "lastName": "Wahba",
        "userName": "MHD1986",
        "email": "best.pid@gmail.com",
        "password": "mhd1234"
    },
    {
        "firstName": "Mauricio",
        "lastName": "Navarro",
        "userName": "Mauro",
        "email": "mauro@gmail.com",
        "password": "mauro46"
    }
]
```

Express

CRUD



implement an API allowing clients to Create, Read, Update, and Delete Expressions. These four functionalities together are known as CRUD, and they form the backbone of many real-life APIs.

13_geoLoction Map

Css Example

css Example
Css Example

---

css Example
Css Example

Understanding ‘this’ in arrow function

Axios , as fetch in js
Npm install –save
Axios.get
Create stream

API-JSON

What is API

https://www.w3schools.com/js/js_json_intro.asp
