Node JS

- ✓ Latest version of NODE JS is 21.7.2.
- ✓ Download node js from https://nodejs.org/en and install it. To check downloaded version type **node -v** in console.
- ✓ The Node.js installer includes the NPM(Node Package Manager). For version check **npm –v**.
- ✓ **NPM is the package manager** for the Node JS platform. It puts modules in place so that node can find them, and manages dependency conflicts intelligently.
- ✓ Node.js is an open source server environment.
- ✓ Node.js allows you to run JavaScript on the server.

REPL

REPL stands for

- **R Read**
- **E Eval**
- **P Print**
- **L Loop**

✓ It represents a computer environment like a Windows console or Unix/Linux shell where a command is entered and the system responds with an output in an interactive mode.

✓ The REPL feature of Node is very useful in experimenting with Node.js codes and to debug JavaScript codes.

It performs the following tasks –

- **Read** − Reads user's input, parses the input into JavaScript data-structure, and stores in memory.
- **Eval** − Takes and evaluates the data structure.
- **Print** − Prints the result.
- **Loop** − Loops the above command until the user presses **ctrl-c** twice.

Starting REPL

REPL can be started by simply running **node** on shell/console without any arguments as follows.

```
$ node
```

You will see the REPL Command prompt > where you can type any Node.js command –

```
$ node
>
```

Simple Expression

Let's try a simple mathematics at the Node.js REPL command prompt –

```
$ node
> 1 + 3
4
> 1 + ( 2 * 3 ) - 4
3
>
```

Use Variables

You can make use variables to store values and print later like any conventional script. If **var** keyword is not used, then the value is stored in the variable and printed. Whereas if **var** keyword is used, then the value is stored but not printed. You can print variables using **console.log()**.

```
$ node
> x = 10
10
> var y = 10
undefined
> x + y
20
> console.log("Hello World")
Hello World
undefined
```

Multiline Expression

Node REPL supports multiline expression similar to JavaScript. Let's check the following do-while loop in action –

```
$ node
> var x = 0
undefined
> do {
  ... x++;
  ... console.log("x: " + x);
  ... }
while ( x < 5 );
x: 1
x: 2
x: 3
x: 4
x: 5
undefined
>
```

... comes automatically when you press Enter after the opening bracket. Node automatically checks the continuity of expressions.

Underscore Variable

You can use underscore **(_)** to get the last result −

```
$ node
> var x = 10
undefined
> var y = 20
undefined
> x + y
30
> var sum = _
undefined
> console.log(sum)
30
undefined
>
```

>.editor // type .editor to enter in editor mode (Block wise execution only)

// Entering editor mode (Ctrl+D to generate output, Ctrl+C to cancel)

```
const fun=(a,b)=>
{console.log("Hello");
return a+b;
}
console.log("Addition is =",fun(10,20));
//Output:
Hello
Addition is = 30
Undefined
```

```
var t=55;
undefined
 do{
... t++
... console.log(t);
... } while(t<=60)
56
57
58
59
60
61
undefined
```

REPL Commands

- **ctrl + c** – terminate the current command.
- **ctrl + c twice** – terminate the Node REPL.
- **ctrl + d** – terminate the Node REPL.
- **Up/Down Keys** – see command history and modify previous commands.
- **tab Keys** – list of current commands.
- **.editor** – enable editor mode to perform tasks
- **.help** – list of all commands.
- **.break** – exit from multiline expression.
- **.clear** – exit from multiline expression.

- **.save *filename*** − save the current Node REPL session to a file. (.save j1.js)
- **.load *filename*** − load file content in current Node REPL session. (.load j1.js)

To remove undefined error

"repl.repl.ignoreUndefined = true"

ignoreUndefined - if set to true, then the repl will not output the return value of command if it's undefined. Defaults to false.

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.

File System (fs) Module

- ✓ The 'fs' module of Node.js implements the File I/O operation.
- ✓ To include the File System module, use the require() method

The syntax for including the fs module in your application:
var fs=require("fs");

- ✓ Methods in the fs module can be synchronous as well as asynchronous.
- ✓ The **Asynchronous** function has a callback function as the last parameter which indicates the completion of the asynchronous function.
- ✓ Node.js developers prefer asynchronous methods over synchronous methods as asynchronous methods never block a program during its execution, whereas the latter does. We will discuss it later.

mkdirSync()

The **fs.mkdirSync()** method is used to create a directory Synchronously.

Syntax:
fs.**mkdirSync**(path, options)

Parameters: This method accept two parameters as mentioned above and described below:

- **path**: The path at which directory is to be created. It can be string, buffer, etc.
- **options**: It is an optional parameter which determines how to create directory like recursively, etc.

renameSync()

The **fs.renameSync()** method is used to synchronously rename a file at the given old path to the given new path. It will overwrite the destination file if it already exists.

Syntax:
fs.**renameSync**(oldPath, newPath)

Property Values:

- **oldPath**: It holds the path of the file that has to be renamed. It can be a string, buffer or URL.
- **newPath**: It holds the new path that the file has to be renamed to. It can be a string, buffer or URL.

writeFileSync()

The fs.**writeFileSync**() is a synchronous method.

The fs.**writeFileSync**() creates a new file if the specified file does not exist.

Syntax:
fs.**writeFileSync**(file, data, options)

readFileSync()

- ✓ The fs.**readFileSync**() method is an inbuilt application programming interface of the fs module which is used to read the file and return its content.
- ✓ In the fs.readFileSync() method, we can synchronously read files, i.e. we are telling node.js to block other parallel processes and do the current file reading process.

Syntax:
fs.**readFileSync**(path, options)

Parameters:

- **path**: It takes the relative path of the text file. The path can be of URL type. The file can also be a file descriptor. If both the files are in the same folder just give the filename in quotes.
- **options**: It is an optional parameter that contains the encoding and flag, the encoding contains data specification. Its default value is null which returns the raw buffer and the flag contains an indication of operations in the file. Its default value is 'r'.

appendFileSync()

- ✓ The fs.**appendFileSync**() method is used to synchronously append the given data to a file.
- ✓ A new file is created if it does not exist. The optional options parameter can be used to modify the behavior of the operation.

Syntax:
fs.appendFileSync(path, data, [options])

Parameters:

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

- **path**: It is a String, Buffer, URL or number that denotes the source filename or file descriptor that will be appended.
- **data**: It is a String or Buffer that denotes the data that has to be appended.

- **options**: It is an string or an object that can be used to specify optional parameters that will affect the output. It has three optional

unlinkSync()

✓ The fs.unlinkSync() method is used to synchronously remove a file or symbolic link from the filesystem.

✓ This function does not work on directories, therefore it is recommended to use fs.rmdir() to remove a directory.

Syntax:
fs.unlinkSync(path)

Parameters:
This method accepts one parameter as mentioned above and described below:

- **path**: It is a string, Buffer or URL which represents the file or symbolic link which has to be removed.

We expect that the **fs.readFile()/fs.readFileSync()** would get the data in text format, i.e., string data type but it returns a buffer object.

In Node.js, the **Buffer.toString()** method is used to decode or convert a buffer to a string, according to the specified character encoding type. Converting a buffer to a string is known as encoding, and converting a string to a buffer is known as decoding.

UTF-8 is the World Wide Web's most common character encoding.

```
fs.readFileSync("node/write.txt",”utf-8”);
```

or

```
data= fs.readFileSync("node/write.txt");
console.log(data.toString());
```

If no encoding is specified, then the raw buffer is returned.

Write node Example with File system methods. (CRUD Operation)

1. **To create folder**
2. **Create one file inside that folder**
3. **Append some data to that file.**

4. **Read data from the file**
5. **Rename that file**
6. **Delete File**

```
var ps=require("fs");
ps.mkdirSync("node");
ps.writeFileSync("node/write.txt","Hello");
ps.appendFileSync("node/write.txt","Hi");
data=ps.readFileSync("node/write.txt");
console.log(data);
console.log(data.toString());  //Or add “utf-8”

//data=ps.readFileSync("node/write.txt",”utf-8”);

ps.renameSync("node/write.txt"," node/readwrite.txt")
ps.unlinkSync("node/readwrite.txt");
```

Read data from file and sort that data in ascending order using .sort() .

```
//string format

var ps=require("fs");
ps.writeFileSync("s1.txt","50 -1 99 100 20 0 56 78 59");
data=ps.readFileSync("S1.txt","utf-8");
data=data.split(" ");
d = data.sort((a,b)=>a-b);
console.log(d);
```

Output:

```
[
  '-1', '0',  '20',
  '50', '56', '59',
  '78', '99', '100'
]
```

```
//integer format

var ps=require("fs");
ps.writeFileSync("s1.txt","10 50 -1 99 100 140 20 0 56 78 59");
data=ps.readFileSync("S1.txt","utf-8");
data=data.split(" ");
```

```
d=data.sort((a,b)=>a-b);
let p=[];
for(i=0;i<d.length;i++){
   p[i]=parseInt(d[i]);

}
console.log(p)
```

Output:

```
[
  -1,  0, 10, 20,  50,
  56, 59, 78, 99, 100,
 140
]
```

Write a node.js script to copy contents of one file to another file. Data should be fetched from Source.txt and insert to destination.txt.

```
        var ps=require("fs");
        ps.writeFileSync("source.txt","ABC");
        data=ps.readFileSync("Source.txt","utf-8");
        ps.writeFileSync("destination.txt",data);
        data1=ps.readFileSync("destination.txt","utf-8");
        console.log(data1);
```

Output:

```
        ABC
```

Asynchronous Programming Using Callbacks

- ✓ Asynchronous programming is an approach to running multiple processes at a time without blocking the other part(s) of the code.
- ✓ There are some cases that code runs (or must run) after something else happens and also not sequentially. This is called asynchronous programming.
- ✓ Callbacks make sure that a function is not going to run before a task is completed but will run right after the task has completed. It helps us develop asynchronous JavaScript code and keeps us safe from problems and errors.
- ✓ In JavaScript, the way to create a callback function is to pass it as a parameter to another function, and then to call it back right after something has happened or some task is completed.

How to create a Callback?

To understand what I've explained above, let me start with a simple example. We want to log a message to the console but it should be there after 3 seconds.

```
const message = function() {
   console.log("This message is shown after 3 seconds");
}
setTimeout(message, 3000);
```

- ✓ There is a built-in method in JavaScript called "setTimeout", which calls a function or evaluates an expression after a given period of time (in milliseconds).
- ✓ So here, the "message" function is being called after 3 seconds have passed. (1 second = 1000 milliseconds)
- ✓ In other words, the message function is being called after something happened (after 3 seconds passed for this example), but not before. So the message function is an example of a callback function.

JavaScript setInterval() Method: The setInterval() method repeats a given function at every given time interval.

JavaScript setTimeout() Method: This method executes a function, after waiting a specified number of milliseconds.

What is an Anonymous Function?

Alternatively, we can define a function directly inside another function, instead of calling it. It will look like this:

```
setTimeout(function() {
   console.log("This message is shown after 3 seconds");
}, 3000);
```

As we can see, the callback function here has no name and a function definition without a name in JavaScript is called as an “anonymous function”. This does exactly the same task as the example above.

Callback as an Arrow Function

If you prefer, you can also write the same callback function as an ES6 arrow function, which is a newer type of function in JavaScript:

```
setTimeout(() => {
   console.log("This message is shown after 3 seconds");
}, 3000);
```

Some callback examples

Display content on browser after 5 seconds

```
<html>
   <head>

   </head>
   <body>
      <p id="id"></p>
      <script>
         setTimeout(myfun,5000);
         function myfun()
         {
            document.getElementById("id").innerHTML="LJU";
         }
      </script>
   </body>
</html>
```

Display addition of two numbers on browser using callback function

```
<html>
<body>
  <p id="demo"></p>
  <script>
    function mydisplay(sum)
    {
      document.getElementById("demo").innerHTML="<b>"+ sum +"</b>";
    }
    function mycals(num1,num2,mycallback)
    {
      sum=num1+num2;
      mycallback(sum);
    }
    mycals(13,15,mydisplay);
  </script>
</body>
</html>
```

Initialize two variables and increment both the variables and display the addition of both the variables at interval of 1 second.

```
<html>
  <body>
    <p id="p1"></p>
    <script>
    function add(a,b)
    {
      obj=document.getElementById("p1");
      obj.innerHTML=(a+b);
    }
    a=2;
    b=5;
    setInterval(
```

```
        function()
        {
           add(++a,++b);
        },1000);
  </script>
  </body>
</html>
```

Write code to increase the font size at interval of 1000 ms and it should stop increasing when the font size reaches to 50px. This task should be performed when you click on "Increase" button on browser. (Default font size = 15px)

```
<html>
  <body>
    <p id="p1"> Hello</p>
    <button onclick="fun1()">Increase</button>
    <script>
    font=15;
    function fun(font)
    {
      document.getElementById("p1").style.fontSize=font+"px";
    }
    function fun1()
    {
       setInterval(
          function()
          {
             if(font<=50)
             {
                fun(font++);
             }
          },1000);
    }
  </script>
  </body>
</html>
```

```
// Without using button
```

```
<html>
  <body>
    <p id="demo" style="color:blue"></p>
    <script>
      size = 15;
      function add() {

        demo.innerHTML = "hello";
        demo.style.color ="red";
        demo.style.fontSize = size;
        if (size <= 50) {
          size++;
        }
      }
      setInterval(add, 1000);
    </script>
  </body>
</html>
```

Write code to perform the tasks as asked below.

- **Add three buttons.**
- **Increase button to increase the fonts. It should stop increasing the fonts when the font size reaches to 200px or stop button is clicked.**
- **Decrease button to decrease the fonts. It should stop decreasing the fonts when the font size reaches to 20px or stop button is clicked.**
- **Stop button to stop increasing or decreasing the fonts.**
- **Increasing/decreasing interval is of 100 ms.**
- **(Default font size = 50px)**

```
<html>
  <body>
    <p id="p1" style="font-size: 50px;"> Hello</p>
    <button onclick="inc()">Increase</button>
    <button onclick="dec()">Decrease</button>
    <button onclick="stop()">stop</button>
    <script>
    font=50;
```

```
    function fun(font)
    {
      document.getElementById('p1').style.fontSize=font+'px';
    }
    function inc()
    {
       test = setInterval(()=>{
            if(font<100){ fun(++font);
            }
         },100);
    }
    function dec()
    {
       test =setInterval(()=>{
            if(font>15){ fun(--font);}
         },100);
    }
    function stop() { clearInterval(test); }
  </script>
  </body>
</html>
```

Asynchronous File system (Non-blocking concept)

- ✓ Asynchronous functions do not block the execution of the program and each command is executed after the previous command even if the previous command has not computed the result.
- ✓ The previous command runs in the background and loads the result once it has finished processing. Thus, these functions are called non-blocking functions.
- ✓ They take a callback function as the last parameter.
- ✓ Asynchronous functions are generally preferred over synchronous functions as they do not block the execution of the program whereas synchronous functions block the execution of the program until it has finished processing. Some of the asynchronous methods of fs module in NodeJS are:

❖ fs.readFile()
❖ fs.writeFile()
❖ fs.appendFile()

fs.writeFile() method

It is used to asynchronously write the specified data to a file. By default, the file would be replaced if it exists. The 'options' parameter can be used to modify the functionality of the method.

Syntax:
fs.writeFile(file, data,options, callback)

Parameters:

This method accepts four parameters as mentioned above and described below:

- **file**: It is a string, Buffer, URL or file description integer that denotes the path of the file where it has to be written. Using a file descriptor will make it behave similar to fs.write() method.
- **data**: It is a string, Buffer, TypedArray or DataView that will be written to the file.
- **callback**: It is the function that would be called when the method is executed.
  - **err**: It is an error that would be thrown if the operation fails.

appendFile() method

The fs.appendFile() method is used to asynchronously append the given data to a file. A new file is created if it does not exist. The options parameter can be used to modify the behavior of the operation.

Syntax:
fs.appendFile(path, data, options, callback)

Parameters:

This method accepts four parameters as mentioned above and described below:

- **path**: It is a String, Buffer, URL or number that denotes the source filename or file descriptor that will be appended to.
- **data**: It is a String or Buffer that denotes the data that has to be appended.
- **options**: It is an string or an object that can be used to specify optional parameters
- **callback**: It is a function that would be called when the method is executed.
    - **err**: It is an error that would be thrown if the method fails.

fs.readFile() method

It is an inbuilt method that is used to read the file. This method read the entire file into the buffer.

Syntax:
fs.readFile(filename, encoding, callback_function)

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

- **filename**: It holds the name of the file to read or the entire path if stored at another location.
- **encoding**: It holds the encoding of the file. Its default value is 'utf8'.
- **callback_function**: It is a callback function that is called after reading of file. It takes two parameters:
    - **err**: If any error occurred.
    - **data**: Contents of the file.

Return Value: It returns the contents/data stored in file or error if any.

Example to understand difference between sync and async.

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

//Synchronous
fs.writeFileSync('test.txt', 'Hello World!')
```

```
console.log('Synchronous Write operation completed.');
console.log("Outside");

//Asynchronous
fs.writeFile('test1.txt', 'Hello World!', function (err) {
   if (err)
      console.log("Error Generated"+err);
   else
    console.log(err)
      console.log('Asynchronous Write operation completed.');
});
console.log("Outside");
```

Output:

```
Synchronous Write operation completed.
Outside

Outside
Asynchronous Write operation completed.
```

Example to understand writeFile, appendFile, readFile, rename, unlink methods.

```
fs = require("fs")
fs.writeFile('test1.txt', 'Hello World!', (err) => {
   if (err) { console.log("Error Generated"+err); }
   else { console.log("Written"); }
});

fs.appendFile('test1.txt', '\nGood Morning!', (err) => {
  if (err) { console.log("Error Generated"+err); }
  else { console.log("Appended"); }
});

fs.readFile('test1.txt',"utf-8", (readErr, data) => {
```

```
   if (readErr) { console.error("Error Generated: "+readErr) }
   else { console.log(data); }
})

fs.rename('test1.txt','test2.txt',() => {console.log("Renamed")})

fs.unlink('test2.txt', unlinkErr => {
  if (unlinkErr) { throw unlinkErr; }
  else { console.log("Deleted")}
});

console.log("Last sentence")
```

Output is dependent on time to be taken to complete the particular task.

Write a Node.js script that asynchronously writes data to a file named 'test1.js'.

If no error occurs during the writing process, the script should then append additional data to the same file.

Finally, it should read the content of the file, including the newly written and appended data, and display it in the console.

Or

Writing data to file, appending data to file and then reading the file data using using ES6 callback.

```
fs = require("fs")
fs.writeFile('test1.txt', 'Hello World!', function (err) {
   if (err) { console.log("Error Generated"+err); }
   else{
      fs.appendFile('test1.txt', '\nGood Morning!', function (err) {
         if (err){ console.log("Error Generated"+err); }
         else{
            fs.readFile('test1.txt',"utf-8", (readErr, data) => {
               if (readErr) console.log("Error Generated: "+readErr)
```

```
            console.log(data);
        });
      }
    });
  }
});
```

OR Use setTimeout to read data

```
fs = require("fs")
fs.writeFile('test1.txt', 'Hello World!', function (err) {
  if (err) console.log("Error Generated"+err);
  fs.appendFile('test1.txt', '\nGood Morning!', function (err) {
    if (err)
      console.log("Error Generated"+err);
  });
});
var rfile = function(){
  setTimeout(function() {
    fs.readFile('test1.txt',"utf-8", (readErr, data) => {
      if (readErr) console.log("Error Generated: "+readErr)
      console.log(data);
      });
  },2000)
}
rfile();
```