

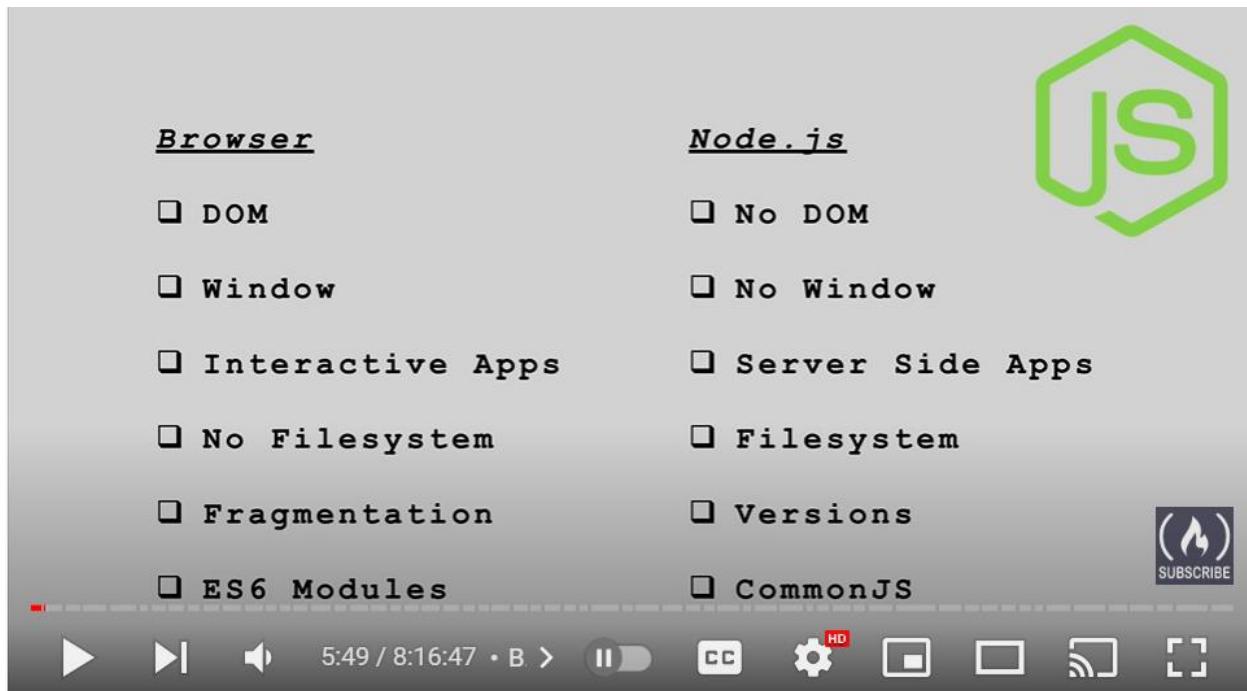
## Node JS

Environment to run JS outside the browser.

Created in 2009.

Built on chrome's V8 Engine.

Every browser has an engine, a tool that compiles our code down to machine code and chrome uses it by the name V8.



Node JS uses Common JS library.

For frontend part, browser does most of the part. Our code is compiled, and browser runs our code.

### How does node evaluate our code?

We have 2 options

1. REPL – Read Eval Print Loop
2. CLI Executable – running our app code in node

REPL is only for playing around but we mostly use CLI.

REPL can be accessed by opening a terminal and type node and then hit enter. Once you have entered the REPL, type const name = "Sai" and click enter.

Now, you can access the name variable which has been defined as 'Sai'.

You can close the REPL by click on Ctrl + C for two times.

## How to run a Node Application in a terminal

Use the command `node <filename>` to run the node app in terminal

Ex: `node app.js`

Ex: `node app` (we can exclude the applications file extension)

The screenshot shows a Visual Studio Code interface. In the center, there is an editor window titled "app.js" containing the following code:

```
const amount = 14;
if (amount > 15) {
  console.log("Large Number");
} else {
  console.log("Small Number");
}
console.log('Hey !! this is my first node application');
```

Below the editor is a terminal window titled "TERMINAL". It shows the output of running the script:

```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Try the new cross-platform PowerShell https://aka.ms/pscore6

PS C:\Users\saikr\Documents\Node JS Tutorial Youtube> node app.js
Large Number
Hey !! this is my first node application
PS C:\Users\saikr\Documents\Node JS Tutorial Youtube> node app.js
Small Number
Hey !! this is my first node application
PS C:\Users\saikr\Documents\Node JS Tutorial Youtube>
```

The terminal window has tabs for "PROBLEMS", "OUTPUT", "DEBUG CONSOLE", and "TERMINAL". The "TERMINAL" tab is currently selected.

## Global Variables in Node

In Vanilla JavaScript, we have access to various global variables like window. With window variable we can have access to many useful functions like querySelector().

There is no window in Node because there is no browser for Node.

Global Variables are nothing but a variable which can be accessed anywhere over the application.

The screenshot shows a Visual Studio Code interface with the following details:

- File Explorer:** Shows a project structure under "NODE JS TUTORIAL YOUTUBE". The "content\\subfolder" directory contains files: test.txt, 1-intro.js, 2-globals.js (selected), 3-modules.js, 4-names.js, 5-utils.js, 6-alternative-flavor.js, 7-mind-grenade.js, 8-os-module.js, and app.js.
- Code Editor:** Displays the content of 2-globals.js. The code uses setinterval() and setTimeout() callbacks to log global variables to the console. It includes comments explaining variables like \_\_dirname, \_\_filename, require, module, process, and console.
- Terminal:** Shows the command "PS C:\Users\saikr\Documents\Node JS Tutorial Youtube>".
- Status Bar:** Shows file statistics: Line 13, Col 20, Spaces: 4, and encoding: UTF-8.

```
// GLOBAL Variables - NO WINDOW
// __dirname - path to current directory
// __filename - file name
// require - function to use modules (CommonJS)
// module - info about current module (file)
// process - info about env where the program is being executed
// console - console is also an global variable which helps to print values on terminal

// there are many more global variables
console.log(__dirname);
console.log(__filename);
setInterval(() => [
  console.log("Hello World !!!"),
], 1000);
setTimeout(() => {
  console.log("Hello Another World !!!");
}, 2000);

// setInterval and setTimeout can also be used in node js
```

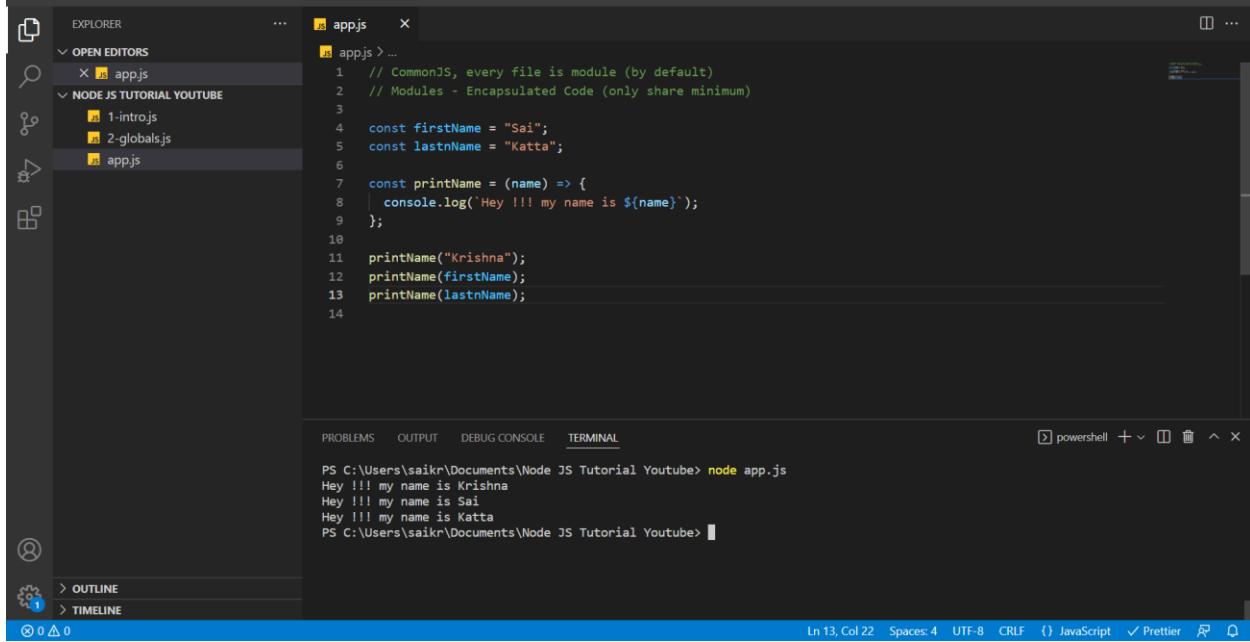
## Modules in Node

Generally, our code will be split into multiple files, but all would be connected to app.js file because we run our entire application runs using one file.

Every File in node is a module.

Modules are Encapsulated code (only share minimum)

## Without the concept of Module



```
// CommonJS, every file is module (by default)
// Modules - Encapsulated Code (only share minimum)

const firstName = "Sai";
const lastName = "Katta";

const printName = (name) => {
  console.log(`Hey !!! my name is ${name}`);
};

printName("Krishna");
printName(firstName);
printName(lastName);
```

PS C:\Users\saikr\Documents\Node JS Tutorial Youtube> node app.js  
Hey !!! my name is Krishna  
Hey !!! my name is Sai  
Hey !!! my name is Katta  
PS C:\Users\saikr\Documents\Node JS Tutorial Youtube>

## With concept of Module

app.js

The screenshot shows the Visual Studio Code interface. The Explorer sidebar on the left lists files: app.js, 5-utilis.js, and 4-names.js. The app.js editor tab is active, displaying the following code:

```
// CommonJS, every file is module (by default)
// Modules - Encapsulated Code (only share minimum)

const names = require("./4-names");
const printName = require("./5-utilis");

// console.log(names);

printName(names.firstName);
printName(names.lastName);
printName("Krishna");
```

The terminal tab at the bottom shows the command `node app.js` being run twice, outputting "Hey !!! my name is Sai", "Hey !!! my name is Katta", and "Hey !!! my name is Krishna".

4-names.js

The screenshot shows the Visual Studio Code interface. The Explorer sidebar on the left lists files: app.js, 5-utilis.js, and 4-names.js. The 4-names.js editor tab is active, displaying the following code:

```
// local variable
const secret = "SUPER SECRET";

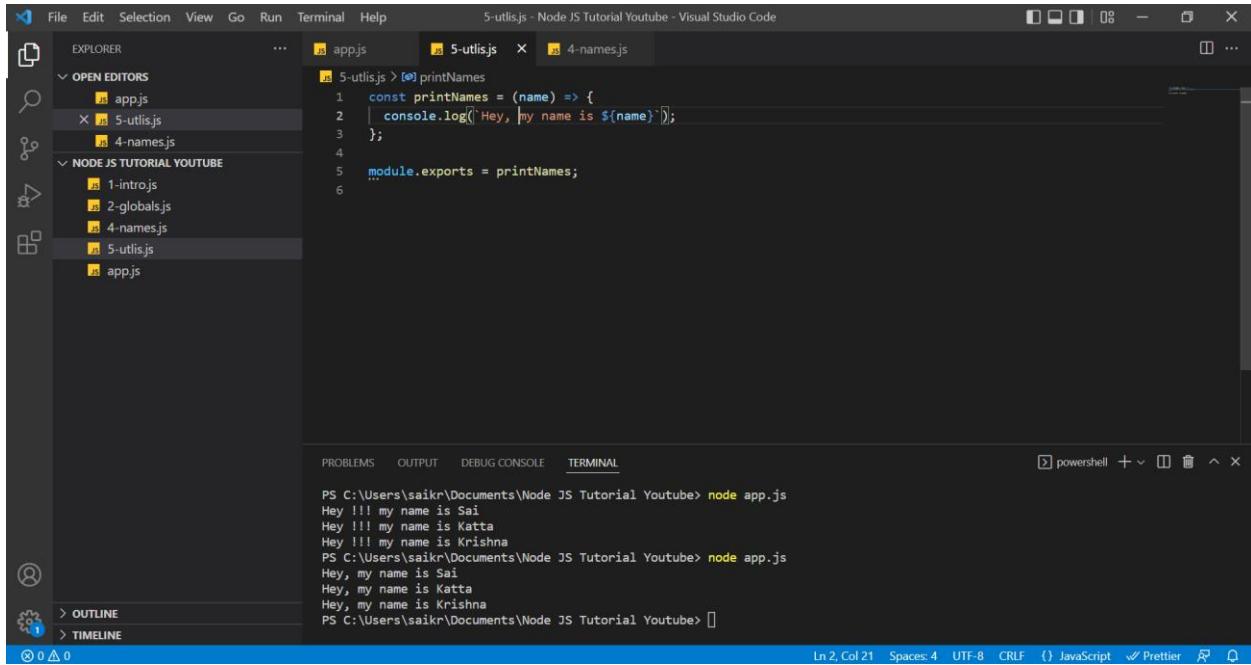
// shared variable
const firstName = "Sai";
const lastName = "Katta";

module.exports = { firstName: "Sai", lastName: "Katta" };

// console.log(module);
```

The terminal tab at the bottom shows the command `node app.js` being run twice, outputting "Hey !!! my name is Sai", "Hey !!! my name is Katta", and "Hey !!! my name is Krishna".

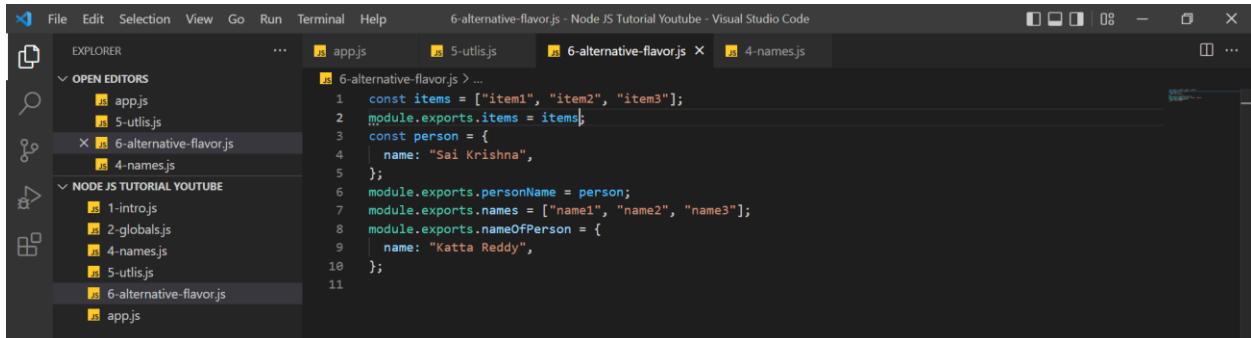
## 5-utils.js



```
File Edit Selection View Go Run Terminal Help 5-utils.js - Node JS Tutorial Youtube - Visual Studio Code
EXPLORER OPEN EDITORS NODE JS TUTORIAL YOUTUBE
app.js 5-utils.js 4-names.js
5-utils.js > [o] printNames
1 const printNames = (name) => {
2   console.log(`Hey, my name is ${name}`);
3 }
4
5 module.exports = printNames;
6

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
PS C:\Users\saikr\Documents\Node JS Tutorial Youtube> node app.js
Hey !!! my name is Sai
Hey !!! my name is Katta
Hey !!! my name is Krishna
PS C:\Users\saikr\Documents\Node JS Tutorial Youtube> node app.js
Hey, my name is Sai
Hey, my name is Katta
Hey, my name is Krishna
PS C:\Users\saikr\Documents\Node JS Tutorial Youtube> []
Ln 2, Col 21  Spaces: 4  UTF-8  CRLF  () JavaScript  ✓ Prettier  ⚙  🔍
```

## 6-alternative-flavors.js



```
File Edit Selection View Go Run Terminal Help 6-alternative-flavor.js - Node JS Tutorial Youtube - Visual Studio Code
EXPLORER OPEN EDITORS NODE JS TUTORIAL YOUTUBE
app.js 5-utils.js 6-alternative-flavor.js 4-names.js
6-alternative-flavor.js > ...
1 const items = ["item1", "item2", "item3"];
2 module.exports.items = items;
3 const person = {
4   name: "Sai Krishna",
5 };
6 module.exports.personName = person;
7 module.exports.names = ["name1", "name2", "name3"];
8 module.exports.nameOfPerson = {
9   name: "Katta Reddy",
10 };
11

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
PS C:\Users\saikr\Documents\Node JS Tutorial Youtube> node app.js
[object Object]
[object Object]
[object Object]
PS C:\Users\saikr\Documents\Node JS Tutorial Youtube> []
Ln 2, Col 21  Spaces: 4  UTF-8  CRLF  () JavaScript  ✓ Prettier  ⚙  🔍
```

We can do multiple exports or else we can use the approach having only module export per file. It depends on the comfort of the person writing code.

In app.js, we can import it only as one single object because in modules we only use export object to export data from one module to another so that is the reason we can have multiple objects like items, personName, names, nameOfPerson from 6-alternative-js inside one object named data of app.js file.

```

File Edit Selection View Go Run Terminal Help
EXPLORER OPEN EDITORS ...
app.js x 5-utiljs x 6-alternative-flavor.js x 4-names.js
app.js > ...
2 // Modules - Encapsulated Code (only share minimum)
3
4 const names = require("./4-names");
5 const printName = require("./5-utiljs");
6
7 // console.log(names);
8
9 // printName(names.firstName);
10 // printName(names.lastName);
11 // printName("Krishna");
12
13 const data = require("./6-alternative-flavor");
14 console.log(data);
15

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
PS C:\Users\saikr\Documents\Node JS Tutorial Youtube> node app.js
{
  items: [ 'item1', 'item2', 'item3' ],
  personName: { name: 'Sai Krishna' },
  names: [ 'name1', 'name2', 'name3' ],
  nameOfPerson: { name: 'Katta Reddy' }
}
PS C:\Users\saikr\Documents\Node JS Tutorial Youtube>

Ln 11, Col 18 Spaces: 4 UTF-8 CRLF () JavaScript ✓ Prettier ⚡ Q

```

Basically, when we import a module in app.js file we invoke that module. Invoking means sometimes we may receive data through exports objects or sometimes we may invoke a function to execute some functionality in that module.

To get a good explanation about that, please have a look at the below screenshots.

```

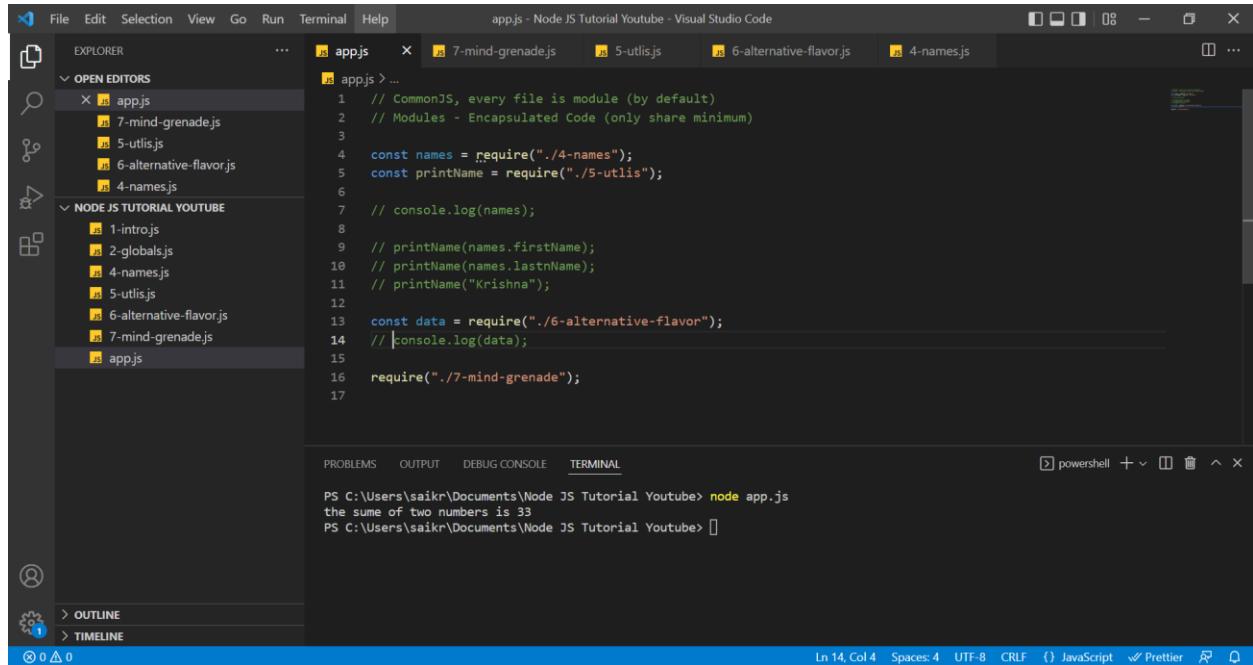
File Edit Selection View Go Run Terminal Help
EXPLORER OPEN EDITORS ...
app.js x 7-mind-grenade.js x 5-utiljs x 6-alternative-flavor.js x 4-names.js
7-mind-grenade.js > ...
1 const num1 = 15;
2 const num2 = 18;
3
4 const addValues = () => {
5   console.log(`the sum of two numbers is ${num1 + num2}`);
6 };
7
8 addValues();
9

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
PS C:\Users\saikr\Documents\Node JS Tutorial Youtube> node app.js
the sum of two numbers is 33
PS C:\Users\saikr\Documents\Node JS Tutorial Youtube>

Ln 9, Col 1 Spaces: 4 UTF-8 CRLF () JavaScript ✓ Prettier ⚡ Q

```

In app.js, when we run node app.js, with the help of **require** global variable we are invoking the module 7-mind-grenade.js, which in turns executes a function named addValues() which calculates sum of two numbers.



The screenshot shows the Visual Studio Code interface with the following details:

- File Explorer:** Shows a tree view of files. Under "OPEN EDITORS", there are files: app.js, 7-mind-grenade.js, 5-utilis.js, 6-alternative-flavor.js, and 4-names.js. Under "NODE JS TUTORIAL YOUTUBE", there are files: 1-intro.js, 2-globals.js, 3-arrays.js, 5-utilis.js, 6-alternative-flavor.js, 7-mind-grenade.js, and app.js.
- Code Editor:** The active file is app.js, containing the following code:

```
// CommonJS, every file is module (by default)
// Modules - Encapsulated Code (only share minimum)

const names = require("./4-names");
const printName = require("./5-utilis");

// console.log(names);

// printName(names.firstName);
// printName(names.lastName);
// printName("Krishna");

const data = require("./6-alternative-flavor");
// |console.log(data);

require("./7-mind-grenade");
```
- Terminal:** Shows the command line output:

```
PS C:\Users\saikr\Documents\Node JS Tutorial Youtube> node app.js
the sum of two numbers is 33
PS C:\Users\saikr\Documents\Node JS Tutorial Youtube>
```
- Status Bar:** Shows "Ln 14, Col 4" and other settings like "Spaces: 4", "UTF-8", "CRLF", "JavaScript", and "Prettier".

This can be great example, since we use a bunch of third-party modules which executes functionalities in background, but we aren't able to see any of those methods on screen. We just extract few methods from the module using **require** global variable and use it in our code.

## Built-in Modules in Node JS

Below are some of the major modules used in Node JS

OS (Operating System Module)

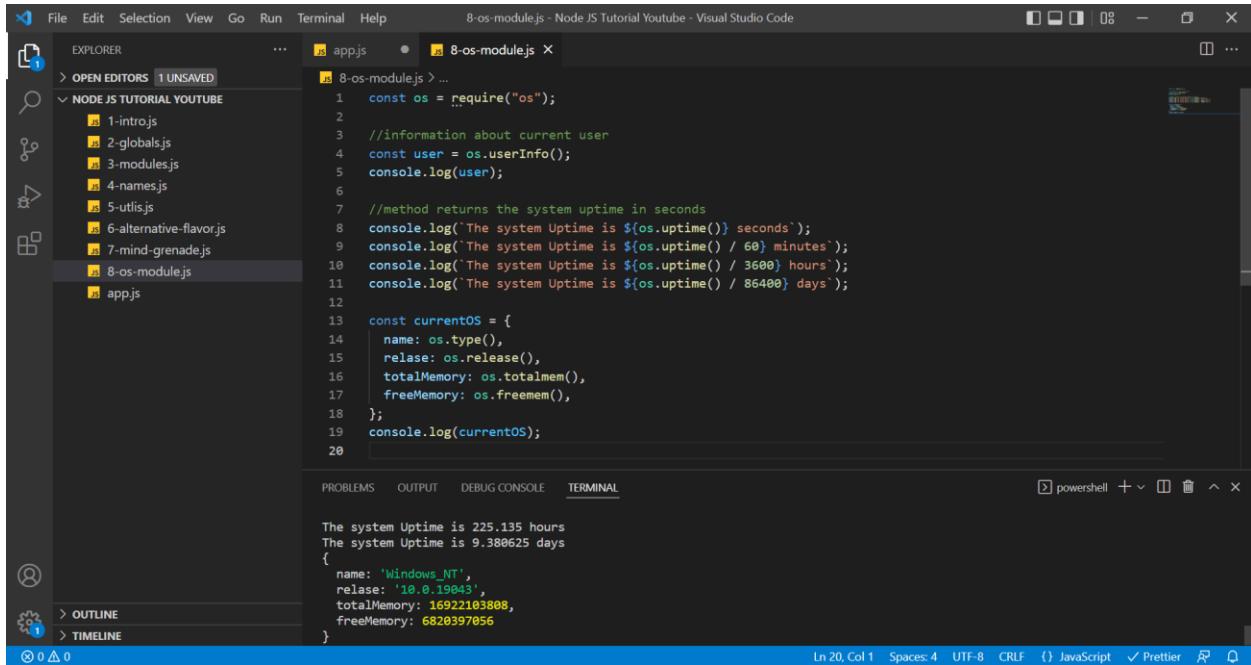
PATH Module

FS (File System Module)

HTTP Module – used to set up our HTTP server

## OS (Operating System Module)

Provides methods and properties that are useful to interact with operating system and as well as server.



The screenshot shows the Visual Studio Code interface with the file `8-os-module.js` open. The code uses the `os` module to log system uptime and details about the current operating system. The terminal output shows the system uptime and the current operating system object.

```
const os = require("os");
//information about current user
const user = os.userInfo();
console.log(user);

//method returns the system uptime in seconds
console.log(`The system Uptime is ${os.uptime()} seconds`);
console.log(`The system Uptime is ${os.uptime() / 60} minutes`);
console.log(`The system Uptime is ${os.uptime() / 3600} hours`);
console.log(`The system Uptime is ${os.uptime() / 86400} days`);

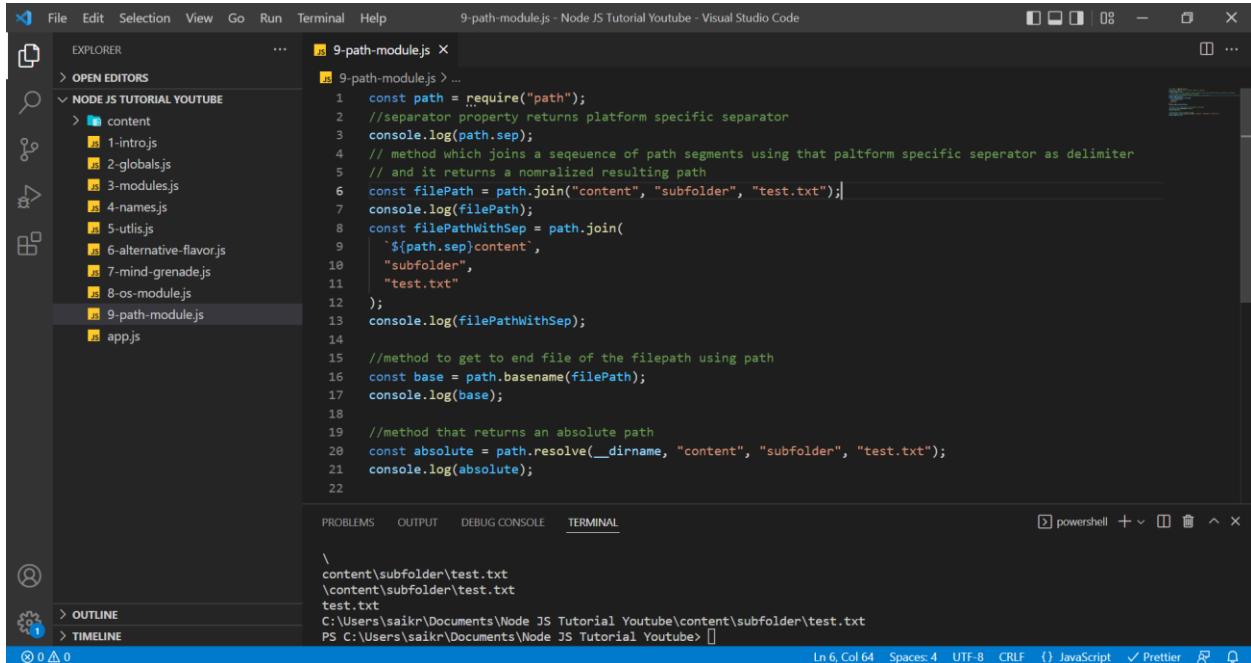
const currentOS = {
  name: os.type(),
  release: os.release(),
  totalMemory: os.totalmem(),
  freeMemory: os.freemem(),
};
console.log(currentOS);

The system Uptime is 225.135 hours
The system Uptime is 9.380625 days
{
  name: 'Windows_NT',
  release: '10.0.19043',
  totalMemory: 16922103888,
  freeMemory: 6820397056
}
```

## PATH Module

With JavaScript in browser, we cannot access File System but in node we can access the File System.

PATH Module allows us to interact with file paths easily.



The screenshot shows the Visual Studio Code interface with the file `9-path-module.js` open. The code demonstrates various methods of the `path` module, such as joining paths, getting the base name, and resolving absolute paths. The terminal output shows the resulting file paths.

```
const path = require("path");
//separator property returns platform specific separator
console.log(path.sep);
// method which joins a sequence of path segments using that platform specific separator as delimiter
// and it returns a normalized resulting path
const filePath = path.join("content", "subfolder", "test.txt");
console.log(filePath);
const filePathWithSep = path.join(
  `${path.sep}content`,
  "subfolder",
  "test.txt"
);
console.log(filePathWithSep);

//method to get to end file of the filepath using path
const base = path.basename(filePath);
console.log(base);

//method that returns an absolute path
const absolute = path.resolve(__dirname, "content", "subfolder", "test.txt");
console.log(absolute);

\content\subfolder\test.txt
\content\subfolder\test.txt
test.txt
C:\Users\saikr\Documents\Node JS Tutorial Youtube\content\subfolder\test.txt
PS C:\Users\saikr\Documents\Node JS Tutorial Youtube> []
```

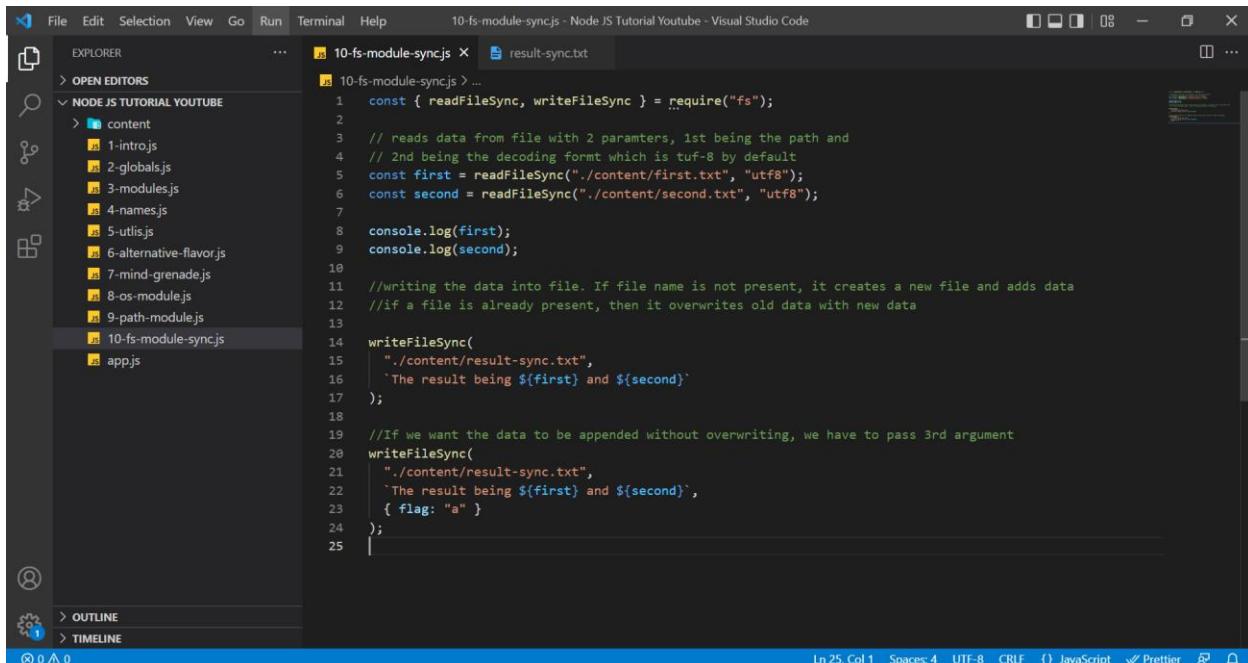
## FS (File System Module)

Accessing File System Module can be done in two ways,

Asynchronously (Which is non-Blocking) and Synchronously (Which is Blocking)

Let us have a look at the **Synchronous** way of reading and writing data into a file.

10-fs-module-sync.js



```
10-fs-module-sync.js - Node JS Tutorial Youtube - Visual Studio Code
```

```
const { readFileSync, writeFileSync } = require("fs");

// reads data from file with 2 parameters, 1st being the path and
// 2nd being the decoding format which is utf-8 by default
const first = readFileSync("./content/first.txt", "utf8");
const second = readFileSync("./content/second.txt", "utf8");

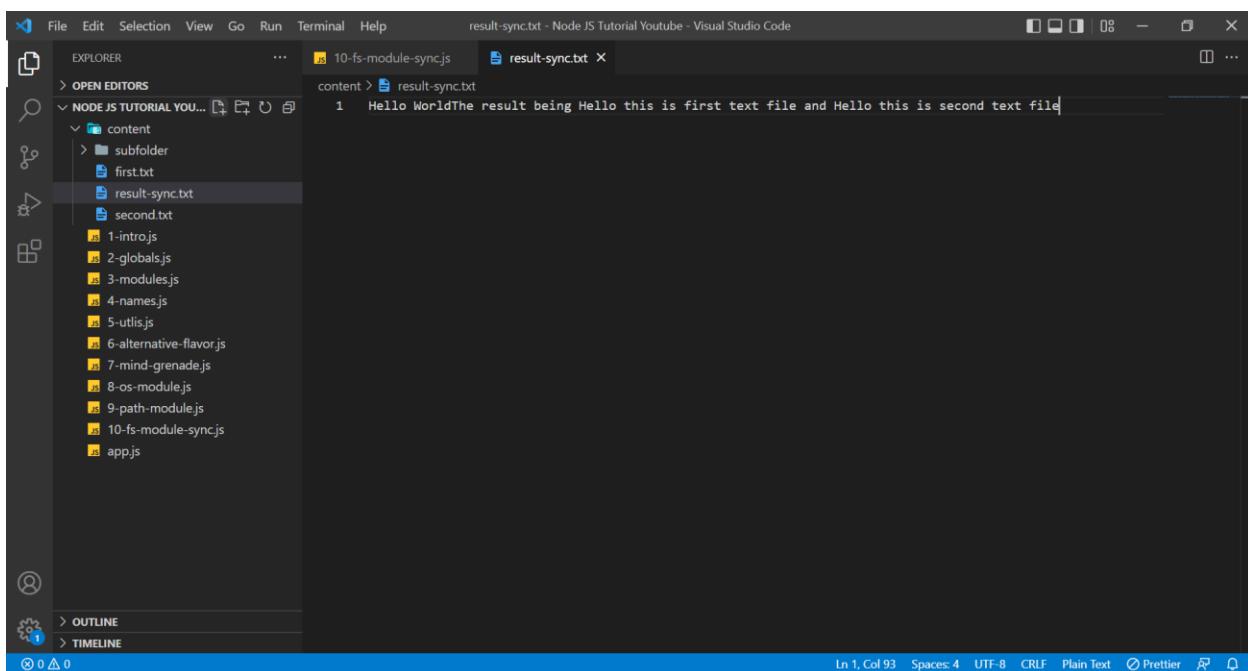
console.log(first);
console.log(second);

//writing the data into file. If file name is not present, it creates a new file and adds data
//if a file is already present, then it overwrites old data with new data

writeFileSync(
    "./content/result-sync.txt",
    `The result being ${first} and ${second}`
);

//If we want the data to be appended without overwriting, we have to pass 3rd argument
writeFileSync(
    "./content/result-sync.txt",
    `The result being ${first} and ${second}`,
    { flag: "a" }
);
```

result-sync.txt



```
result-sync.txt - Node JS Tutorial Youtube - Visual Studio Code
```

```
content > result-sync.txt
```

```
Hello WorldThe result being Hello this is first text file and Hello this is second text file
```

If you try to read data from a non-existed file, then node will throw an error. Here we are taking files which are not existing. Here the node is throwing an error.

File Edit Selection View Go Run Terminal Help 10-fs-module-sync.js - Node JS Tutorial Youtube - Visual Studio Code

EXPLORER OPEN EDITORS NODE JS TUTORIAL YOUTUBE content subfolder first.txt result-sync.txt second.txt 1-introjs 2-global.js 3-modules.js 4-names.js 5-util.js 6-alternative-flavor.js 7-mind-grenade.js 8-os-module.js 9-path-module.js 10-fs-module-sync.js app.js

10-fs-module-sync.js

```
3 // reads data from file with 2 parameters, 1st being the path and
4 // 2nd being the decoding format which is utf-8 by default
5 const first = readFileSync("./content/firsst.txt", "utf8");
6 const second = readFileSync("./content/seccond.txt", "utf8");
7
8 console.log(first);
9 console.log(second);
10
11 //writing the data into file. If file name is not present, it creates a new file and adds data
12 //if a file is already present, then it overwrites old data with new data
13
14 writeFileSync(
15   "./content/result-sync.txt",
16   `The result being ${first} and ${second}`
17 );
18
19 //If we want the data to be appended without overwriting, we have to pass 3rd argument
20 writeFileSync(
21   "./content/result-sync.txt",
22   "This is appended text"
23 );
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

Windows PowerShell  
PS C:\Users\saikr\Documents\Node JS Tutorial Youtube> node 10-fs-module-sync  
internal/fs/utils.js:314  
throw err;  
^

Error: ENOENT: no such file or directory, open './content/firsst.txt'  
at Object.openSync (fs.js:498:3)  
at readFileSync (fs.js:394:35)

Ln 6, Col 62 (122 selected) Spaces: 2 UTF-8 CRLF {} JavaScript ✓ Prettier

Now we are reading two files and combining their data and writing that data into a new text file result-async.txt. The result of readfile is undefined. This is also creating a callback hell.

The screenshot shows a Visual Studio Code interface with the following details:

- File Explorer:** Shows a tree view of files under "NODE JS TUTORIAL YOUTUBE". Files listed include: content, first.txt, result-async.txt, result-sync.txt, second.txt, 1-intro.js, 2-globals.js, 3-modules.js, 4-names.js, 5-utilis.js, 6-alternative-flavor.js, 7-mind-grenade.js, 8-os-module.js, 9-path-module.js, 10-ts-module-sync.js, and app.js.
- Code Editor:** The active file is "app.js". The code reads two files ("first.txt" and "second.txt") using asynchronous file operations and writes their contents to a new file ("result-async.txt").
- Terminal:** At the bottom, the terminal window shows the command "node app.js" being run, followed by the output: "Hello this is first text file" and "Hello this is second text file".

## result-async.txt

The screenshot shows the Visual Studio Code interface. The Explorer sidebar on the left lists files and folders under 'NODE JS TUTORIAL YOUTUBE'. The 'content' folder contains 'first.txt', 'result-async.txt', 'result-sync.txt', and 'second.txt'. The 'OPEN EDITORS' tab shows 'app.js' and 'result-async.txt'. The 'result-async.txt' editor has the following content:

```
Hello this is first text file and Hello this is first text file
```

The Terminal tab at the bottom shows the command 'node app.js' being run, with the output:

```
Hello this is first text file
Hello this is second text file
undefined
```

When we try to read the data and couldn't find the file.

The screenshot shows the Visual Studio Code interface. The Explorer sidebar on the left lists files and folders under 'NODE JS TUTORIAL YOUTUBE'. The 'OPEN EDITORS' tab shows 'app.js' and 'result-async.txt'. The 'app.js' editor contains the following code:

```
const fs = require('fs');
const readline = require('readline');

const rl = readline.createInterface({
  input: fs.createReadStream('./content/first.txt'),
  output: fs.createWriteStream('./content/result-async.txt')
});

rl.on('line', (line) => {
  console.log(`Line: ${line}`);
});

rl.on('close', () => {
  process.exit(0);
});
```

The Terminal tab at the bottom shows the command 'node app.js' being run, with the output:

```
Hello this is first text file
[Error: ENOENT: no such file or directory, open 'C:\Users\saikr\Documents\Node JS Tutorial Youtube\content\seccond.txt']
  errno: -4058,
  code: 'ENOENT',
  syscall: 'open',
  path: 'C:\Users\saikr\Documents\Node JS Tutorial Youtube\content\seccond.txt'
]
```

result-async.txt (result will be only from first result variable which is “Hello this is first text file”)

The screenshot shows the Visual Studio Code interface with the following details:

- File Explorer:** Shows a folder structure under "NODE JS TUTORIAL YOUTUBE" containing "content", "first.txt", "result-async.txt", "result-sync.txt", and "second.txt". Below "content" are subfolders "1-intro.js" through "10-fs-module-sync.js" and "app.js".
- Terminal:** Displays the command "node app.js" and its output:

```
PS C:\Users\saikr\Documents\Node JS Tutorial Youtube> node app.js
Hello this is first text file
[Error: ENOENT: no such file or directory, open 'C:\Users\saikr\Documents\Node JS Tutorial Youtube\content\second.txt' {
  errno: -4058,
  code: 'ENOENT',
  syscall: 'open',
  path: 'C:\Users\saikr\Documents\Node JS Tutorial Youtube\content\second.txt'
}]
undefined
PS C:\Users\saikr\Documents\Node JS Tutorial Youtube>
```
- Status Bar:** Shows "Ln 1, Col 64" and other standard status bar information.

## Sync vs Async

In Synchronous way, JavaScript executes code in line-by-line fashion, which means it executes line after line even if task 2 takes more time, it will wait until the task 2 execution is complete and then moves on to task 3.

The screenshot shows the Visual Studio Code interface with the following details:

- File Explorer:** Shows a folder structure under "NODE JS TUTORIAL YOUTUBE" containing "content", "first.txt", "result-async.txt", "result-sync.txt", and "second.txt". Below "content" are subfolders "1-intro.js" through "10-fs-module-sync.js" and "app.js".
- Terminal:** Displays the command "node 10-fs-module-sync.js" and its output:

```
PS C:\Users\saikr\Documents\Node JS Tutorial Youtube> node 10-fs-module-sync.js
10-fs-module-sync.js >
1  const { readFileSync, writeFileSync } = require("fs");
2  console.log("Start");
3
4 // reads data from file with 2 parameters, 1st being the path and
5 // 2nd being the decoding format which is tuf-8 by default
6 const first = readFileSync("./content/first.txt", "utf8");
7 const second = readFileSync("./content/second.txt", "utf8");
8
9 //writing the data into file. If file name is not present, it creates a new file and adds data
10 //if a file is already present, then it overwrites old data with new data
11
12 writeFileSync(
13   "./content/result-sync.txt",
14   `The result being ${first} and ${second}`);
15
16 //If we want the data to be appended without overwriting, we have to pass 3rd argument
17 writeFileSync(
18   "./content/result-sync.txt",
19   `The result being ${first} and ${second}`,
20   { flag: "a" });
21
22
23 console.log("Done with this task");
24 console.log("Starting next task");
25
```
- Status Bar:** Shows "Ln 23, Col 36" and other standard status bar information.

In Asynchronous way, JavaScript executes line-by-line fashion but if it sees any task (assume task 2) taking time it will offload the task2 and move on to another task but in the meanwhile the task 2 runs in the background and once it is done it will provide the results.

In Asynchronous way, we handle the tasks using the callback function but if we use too many callback functions it will create a Callback Hell and hence, we use Promises or Async and Await for asynchronous.

```

OPEN EDITORS 1 UNSAVED
EXPLORER 11-fs-module-async.js 10-fs-module-sync.js 11-fs-module-async.js
NODE JS TUTORIAL YOUTUBE
content
1-intro.js
2-global.js
3-modules.js
4-names.js
5-util.js
6-alternative-flavor.js
7-mind-grenade.js
8-os-module.js
9-path-module.js
10-fs-module-sync.js
11-fs-module-async.js
app.js

11-fs-module-async.js > ...
1 const { readfile, writefile } = require("fs");
2 console.log("Start");
3 readfile("./content/first.txt", "utf8", (err, result) => {
4   if (err) {
5     console.log(err);
6     return;
7   }
8   const first = result;
9   // console.log(result);
10  readfile("./content/second.txt", "utf8", (err, result) => {
11    if (err) {
12      console.log(err);
13      return;
14    }
15    // console.log(result);
16  });
17  const second = result;
18  console.log("Done with this task");
19  writefile(
20    "./content/result-async.txt",
21    `${first} and ${second}`,
22    (err, result) => {
23      if (err) {
24        console.log(err);
25        return;
26      }
27      // console.log(result);
28    }
29  );
30 });
31 console.log("Starting next task");

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

PS C:\Users\saikr\Documents\Node JS Tutorial Youtube> node 11-fs-module-async  
Start  
Starting next task  
Done with this task  
PS C:\Users\saikr\Documents\Node JS Tutorial Youtube>

## HTTP Module

HTTP Module is used to setup a web server.

```

OPEN EDITORS
EXPLORER app.js
NODE JS TUTORIAL YOUTUBE
content
1-intro.js
2-global.js
3-modules.js
4-names.js
5-util.js
6-alternative-flavor.js
7-mind-grenade.js
8-os-module.js
9-path-module.js
10-fs-module-sync.js
11-fs-module-async.js
app.js

app.js > (e) server > http.createServer() callback
1 //Instead of using write and end methods, we can use end method itself
2 //res.end("Hi! welcome to our home page");
3 res.write("Hi! welcome to our home page");
4 res.end();
5 return;
6 }
7 if (req.url === "/about") {
8   //Instead of using write and end methods, we can use end method itself
9   //res.end("Here is our Short History");
10  res.write("Here is our Short History");
11  res.end();
12  return;
13 }
14 //instead of using write and end methods, we can use end method itself
15 //res.end("<h1>Oops!!</h1>");
16 //p>The page you are looking for is not available</p>
17 //a href="/">Back to Home Page</a>
18 res.write( "<h1>Oops!!</h1>" );
19 <p>The page you are looking for is not available</p>
20 <a href="/">Back to Home Page</a>;
21 res.end();
22 }
23 //listen method is used to listen the server at particular port number
24 server.listen(5000);
25
26
27
28
29
30
31

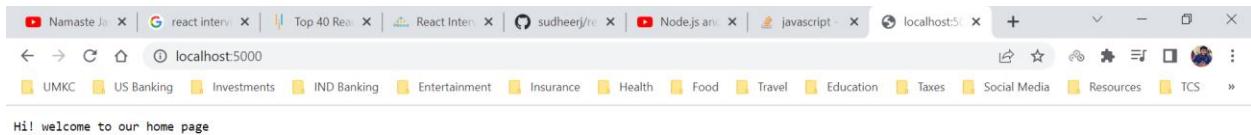
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

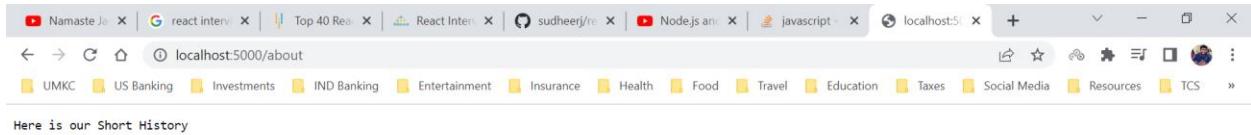
PS C:\Users\saikr\Documents\Node JS Tutorial Youtube> node app

Web Server is started when we run node app.js command in the terminal.

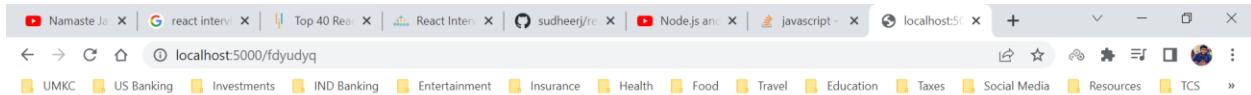
## Home Page on localhost:5000



## About Page on localhost:5000/about



## Error Page on localhost:5000/fdyudyq

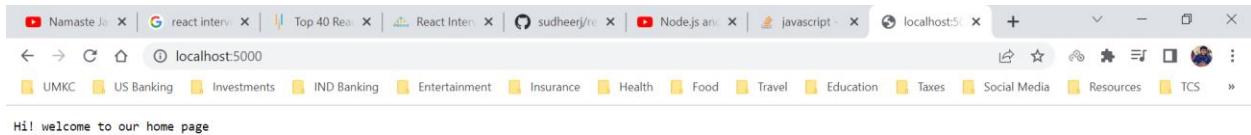


**Oops!!!**

**The page you are looking for is not available**

**Back to Home Page**

## On Click of Home Page Link, we will be redirected to Home Page



Hi! welcome to our home page

## NPM Information

When we install node, we automatically install NPM (Node Package Manager). NPM enables to do 3 things

1. Reuse our code in another project.
2. Use code written by other developers using a npm command.
3. Share our solutions with other developers as well.

NPM is hosted on [npmjs.com](https://npmjs.com), here we can find useful utility functions to full blown frameworks and libraries.

Example: React is available in npm using ***npx create-react-app <app-name>***

A typical node project will have more than few packages installed as dependencies.

NPM calls the reusable code, basically a package. A package is a folder that contains a JavaScript code. Package can be also called as dependency and module as well.

There is no quality control in npm registry. Anyone can publish anything on the registry. A good indication for good and secure package is the number of weekly downloads.

## NPM in Node

Since NPM is already installed with node, In Node JS environment we have access to NPM global variable `npm`.

`npm - -v`

`package.json` - manifest file (stores important info about project/package)

3 ways to create `package.json` file

1. manual approach (create `package.json` in the root, create properties etc.)
2. `npm init` (step by step, press enter to skip)
3. `npm init -y` (everything default)

The screenshot shows the Visual Studio Code interface. The left sidebar has 'OPEN EDITORS' expanded, showing 'NODE JS TUTORIAL YOUTUBE' with several sub-folders like 'content', 'node\_modules', 'bootstrap', 'lodash', etc., and files like '1-intro.js' through '12-http-module.js', 'app.js', 'package-lock.json', and 'package.json'. The main editor area has two tabs: 'app.js' and 'package.json'. The 'app.js' tab contains the following code:

```
// npm - global command, comes with node
// npm --v
//
// local dependency - use it only in this particular project
// npm i <packagename>
//
// global dependency - use it in any project
// npm install -g <packagename>
// sudo npm install -g <packagename> (mac) (In Mac, most likely the password will be asked)
//
// package.json - manifest file (stores important info about project/package)
// 3 ways to create package.json file
// 1. manual approach (create package.json in the root, create properties etc)
// 2. npm init (step by step, press enter to skip)
// 3. npm init -y (everything default)
```

The 'package.json' tab is also visible. Below the editor is a status bar with 'powershell' and other icons. The bottom right corner shows 'Ln 16, Col 1' and other settings.

If you wish to publish this NPM Registry, then select a unique name for package which is not available in the NPM registry.

The screenshot shows the Visual Studio Code interface with the 'package.json' file open. The code in the editor is:

```
{
  "name": "nodejs-tutorial",
  "version": "1.0.0",
  "description": "",
  "main": "1-intro.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "sai krishna",
  "license": "ISC"
}
```

The 'package.json' tab is active. Below the editor is a status bar with 'powershell' and other icons. The bottom right corner shows 'Ln 1, Col 1' and other settings. A message 'Is this OK? (yes)' is displayed in the terminal.

We will try to install first package into our project. The lodash is a utility library. Now we can see dependencies object inside the package.json file. We can also see node\_modules folder is installed with lodash dependency.

The screenshot shows the Visual Studio Code interface. In the Explorer sidebar, there is a 'NODE JS TUTORIAL YOUTUBE' folder containing several files like 'content', 'node\_modules', '1-intro.js', etc. The 'package.json' file is open in the editor. Its content is:

```
1 {
  "name": "nodejs-tutorial",
  "version": "1.0.0",
  "description": "",
  "main": "1-intro.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "sai krishna",
  "license": "ISC",
  "dependencies": {
    "lodash": "^4.17.21"
  }
}
```

In the Terminal tab, the command 'npm i lodash' is run, and the output shows:

```
PS C:\Users\saikr\Documents\Node JS Tutorial Youtube> npm i lodash
npm notice created a lockfile as package-lock.json. You should commit this file.
npm WARN nodejs-tutorial@1.0.0 No description
npm WARN nodejs-tutorial@1.0.0 No repository field.

+ lodash@4.17.21
added 1 package from 2 contributors and audited 1 package in 2.389s
found 0 vulnerabilities
```

The status bar at the bottom indicates: Line 15, Col 1 | Spaces: 2 | UTF-8 | LF | {} JSON | ✓ Prettier

node\_modules is the folder where all the dependencies are stored.

Installed bootstrap library as well.

The screenshot shows the Visual Studio Code interface. In the Explorer sidebar, there is a 'NODE JS TUTORIAL YOUTUBE' folder containing 'content', 'node\_modules', 'bootstrap', and 'lodash'. The 'package.json' file is open in the editor. Its content is:

```
1 {
  "name": "nodejs-tutorial",
  "version": "1.0.0",
  "description": "",
  "main": "1-intro.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "sai krishna",
  "license": "ISC",
  "dependencies": {
    "bootstrap": "^5.1.3",
    "lodash": "^4.17.21"
  }
}
```

In the Terminal tab, the command 'npm i bootstrap' is run, and the output shows:

```
PS C:\Users\saikr\Documents\Node JS Tutorial Youtube> npm i bootstrap
npm WARN bootstrap@5.1.3 requires a peer of @popperjs/core@^2.10.2 but none is installed. You must install peer dependencies yourself.
npm WARN nodejs-tutorial@1.0.0 No description
npm WARN nodejs-tutorial@1.0.0 No repository field.

+ bootstrap@5.1.3
added 1 package from 2 contributors and audited 2 packages in 2.016s

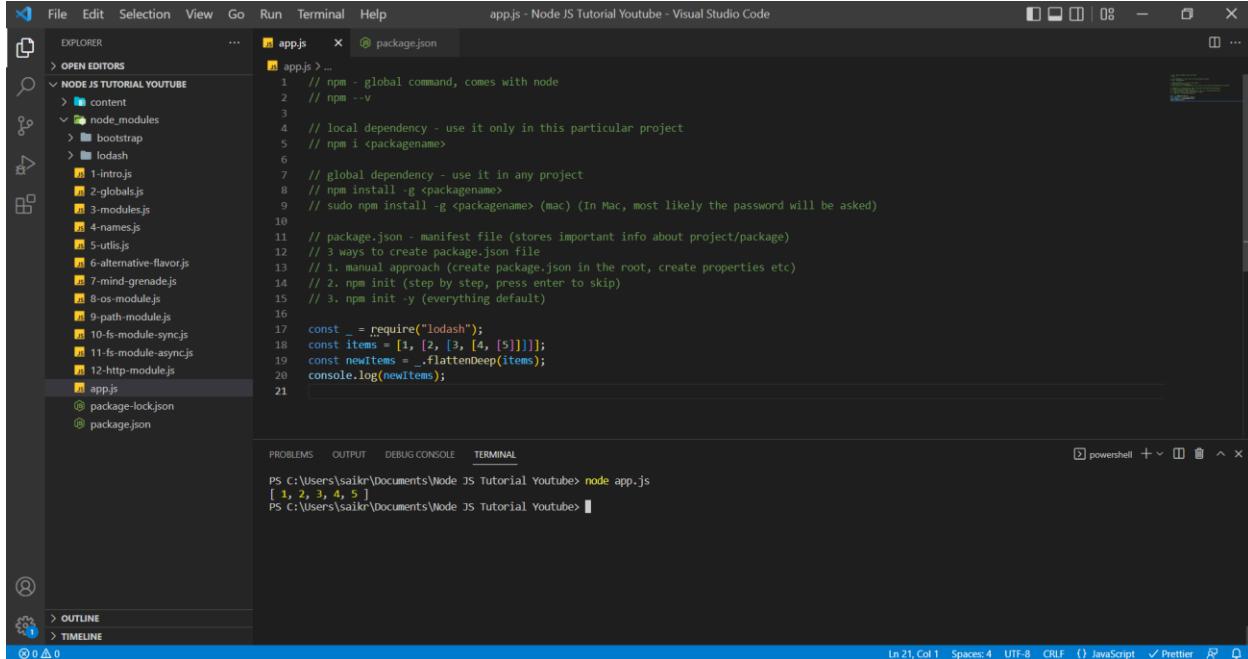
1 package is looking for funding
  run 'npm fund' for details

found 0 vulnerabilities
```

The status bar at the bottom indicates: Line 15, Col 2 | Spaces: 2 | UTF-8 | LF | {} JSON | ✓ Prettier

In case of external dependencies, we need to first install them through NPM if we don't install then we won't be able to find it. In case of internal dependencies like in-built modules, we don't need to install anything.

### Using loadash in our project



The screenshot shows the Visual Studio Code interface. The code editor displays a file named 'app.js' with the following content:

```
// npm - global command, comes with node
// npm --v
//
// local dependency - use it only in this particular project
// npm i <packagename>
//
// global dependency - use it in any project
// npm install -g <packagename> (mac) (In Mac, most likely the password will be asked)
//
// package.json - manifest file (stores important info about project/package)
// 3 ways to create package.json file
// 1. manual approach (create package.json in the root, create properties etc)
// 2. npm init (step by step, press enter to skip)
// 3. npm init -y (everything default)

const _ = require("lodash");
const items = [1, [2, [3, [4, [5]]]]];
const newItems = _.flattenDeep(items);
console.log(newItems);
```

The terminal below shows the output of running the script:

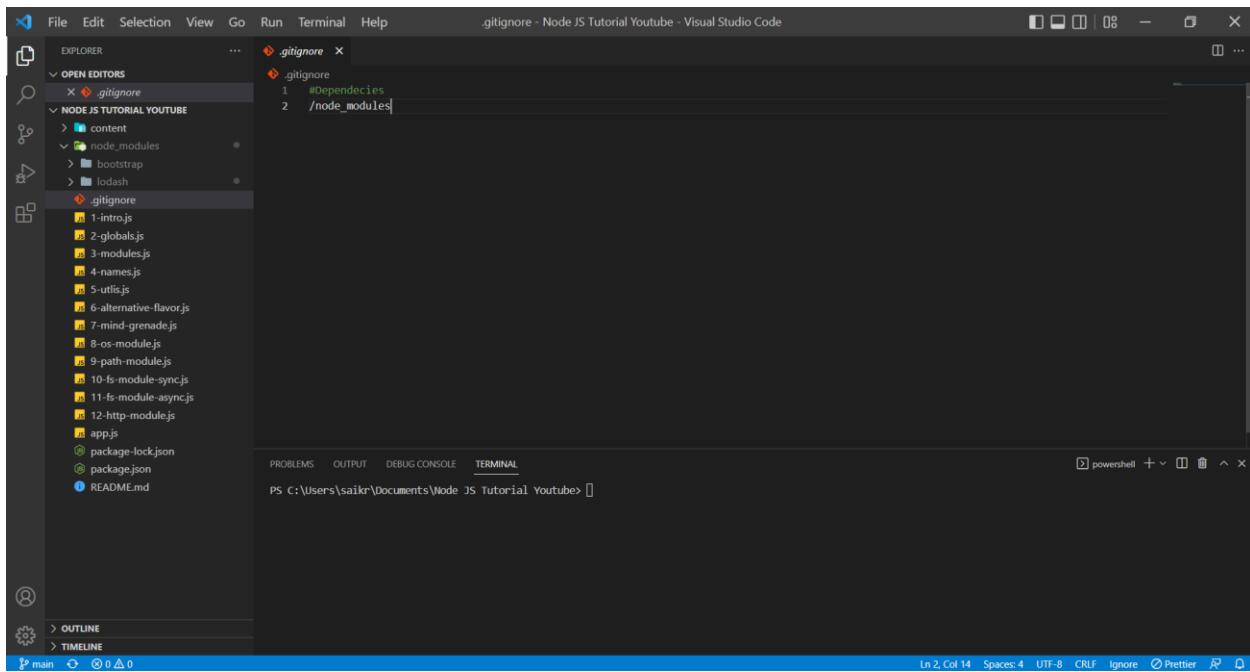
```
PS C:\Users\saikr\Documents\Node JS Tutorial Youtube> node app.js
[ 1, 2, 3, 4, 5 ]
PS C:\Users\saikr\Documents\Node JS Tutorial Youtube>
```

### Sharing Code with Other Developers

We can share the code using the github repository. We cannot upload node\_modules folder into the github repository. Hence, we create a file named .gitignore which will ignore the folders that we mention in the file.

It is very important to have package.json file because, we can take a repository from github and then use the command npm install to install all the dependencies into our project.

.gitignore file as mentioned earlier.



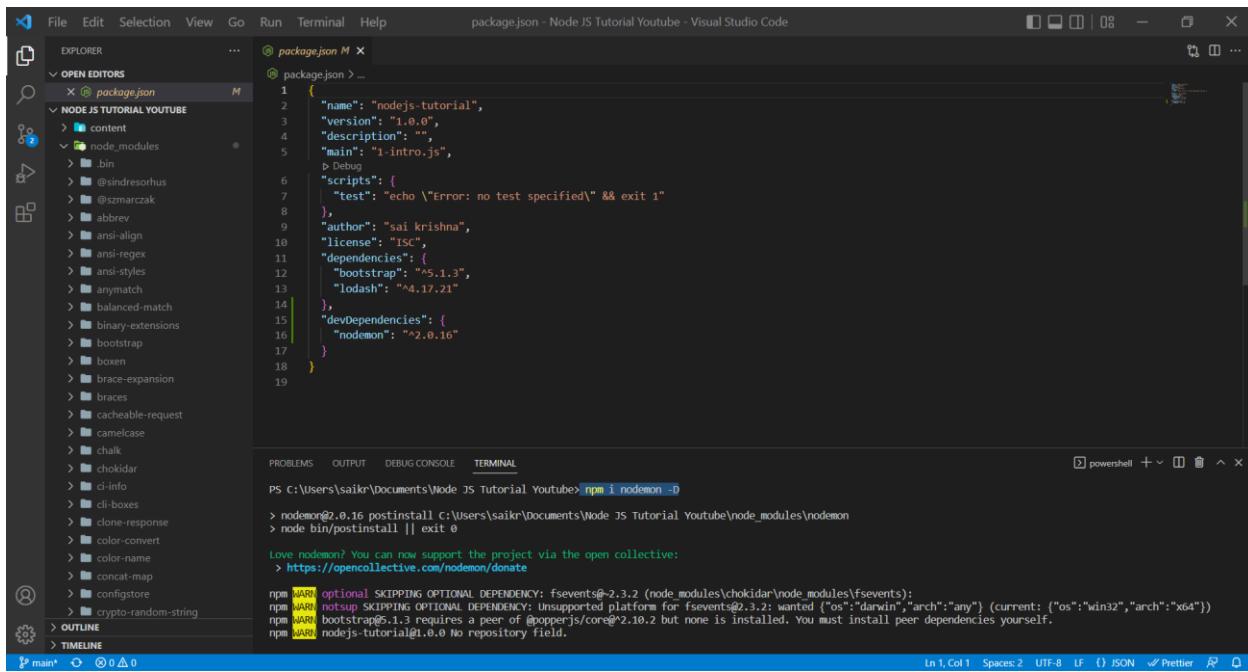
## Nodemon

Nodemon is one of the packages of the npm. It is going to watch our file system and restart the application for us. In that way we don't have to type node <filename>. We can install it as dependency but here we are installing it as a dev dependency.

Command to install a package as dev dependency

***npm i <package-name> -D or npm i <package-name> --save-dev***

## Nodemon as dev dependency In package.json file



The screenshot shows the Visual Studio Code interface with the following details:

- File Explorer:** Shows the project structure under "NODE JS TUTORIAL YOUTUBE".
- Editor:** The "package.json" file is open, displaying the following JSON code:

```
1 {
2   "name": "nodejs-tutorial",
3   "version": "1.0.0",
4   "description": "",
5   "main": "1-intro.js",
6   "scripts": {
7     "test": "echo \\"Warning: no test specified\\\" && exit 1"
8   },
9   "author": "sai krishna",
10  "license": "ISC",
11  "dependencies": {
12    "bootstrap": "^5.1.3",
13    "lodash": "4.17.21"
14  },
15  "devDependencies": {
16    "nodemon": "^2.0.16"
17  }
18}
```

- Terminal:** Shows the command `npm i nodemon -D` being run in the terminal, followed by the output of the command.

```
PS C:\Users\saikr\Documents\Node JS Tutorial Youtube> npm i nodemon -D
> nodemon@2.0.16 postinstall C:\Users\saikr\Documents\Node JS Tutorial Youtube\node_modules\nodemon
> node bin/postinstall || exit 0

Love nodemon? You can now support the project via the open collective:
> https://opencollective.com/nodemon/donate

npm WARN optional SKIPPING OPTIONAL DEPENDENCY: fsevents@2.3.2 (node_modules\chokidar\node_modules\fsevents):
npm WARN notsup SKIPPING OPTIONAL DEPENDENCY: Unsupported platform for fsevents@2.3.2: wanted {"os":"darwin","arch":"any"} (current: {"os":"win32","arch":"x64"})
npm WARN bootstrap@5.1.3 requires a peer of @popperjs/core@^2.10.2 but none is installed. You must install peer dependencies yourself.
npm WARN nodejs-tutorial@1.0.0 No repository field.
```

- Status Bar:** Shows the current file is "package.json - Node JS Tutorial Youtube - Visual Studio Code".

In production we may not use the Nodemon, but Heroku and other services will be using other serious packages but in development stage this is an option.

Basically, devDependencies is nothing but the dependencies that we are using for the development purposes but once we get into production, we only use the dependencies used by our application.

Inside the scripts object of the package.json file we just setup some commands which we want to run. For example, we setup start which indeed is a replacement for command ***node <file-name>***.

We have run command **npm start** which in turn is starting the node application.

For some commands, we just need to use npm <name-of-command> like **npm start**.

With the **start** command which is nothing but **node app.js**, we are stopping the application after executing the command.

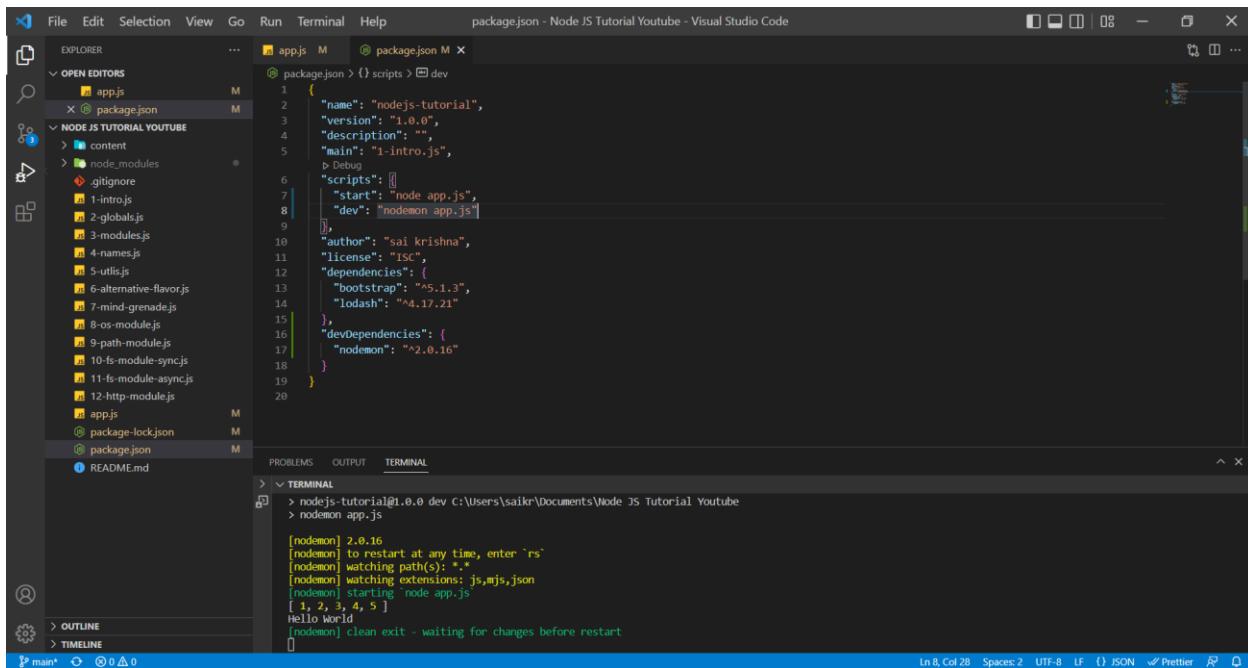
The screenshot shows the Visual Studio Code interface. In the Explorer sidebar, there are several files and folders: .gitignore, 1-intro.js, 2-globals.js, 3-modules.js, 4-names.js, 5-utils.js, 6-alternative-flavor.js, 7-mind-grenade.js, 8-os-module.js, 9-path-module.js, 10-fs-module-sync.js, 11-fs-module-async.js, 12-http-module.js, app.js, package-lock.json, package.json, and README.md. The package.json file is open in the editor, showing the following content:

```
1  {
2   "name": "nodejs-tutorial",
3   "version": "1.0.0",
4   "description": "",
5   "main": "1-intro.js",
6   "scripts": {
7     "start": "node app.js"
8   },
9   "author": "sai krishna",
10  "license": "ISC",
11  "dependencies": {
12    "bootstrap": "^5.1.3",
13    "lodash": "4.17.21"
14  },
15  "devDependencies": {
16    "nodemon": "^2.0.16"
17  }
18}
```

In the terminal tab, the command `npm start` is being run, and the output shows the application is running and printing the numbers 1 through 5 to the console.

```
PS C:\Users\saikr\Documents\Node JS Tutorial Youtube> npm start
> nodejs-tutorial@1.0.0 start C:\Users\saikr\Documents\Node JS Tutorial Youtube
> node app.js
[ 1, 2, 3, 4, 5 ]
PS C:\Users\saikr\Documents\Node JS Tutorial Youtube>
```

For some other commands, we need to run `npm run <name-of-command>` like `npm run dev`.



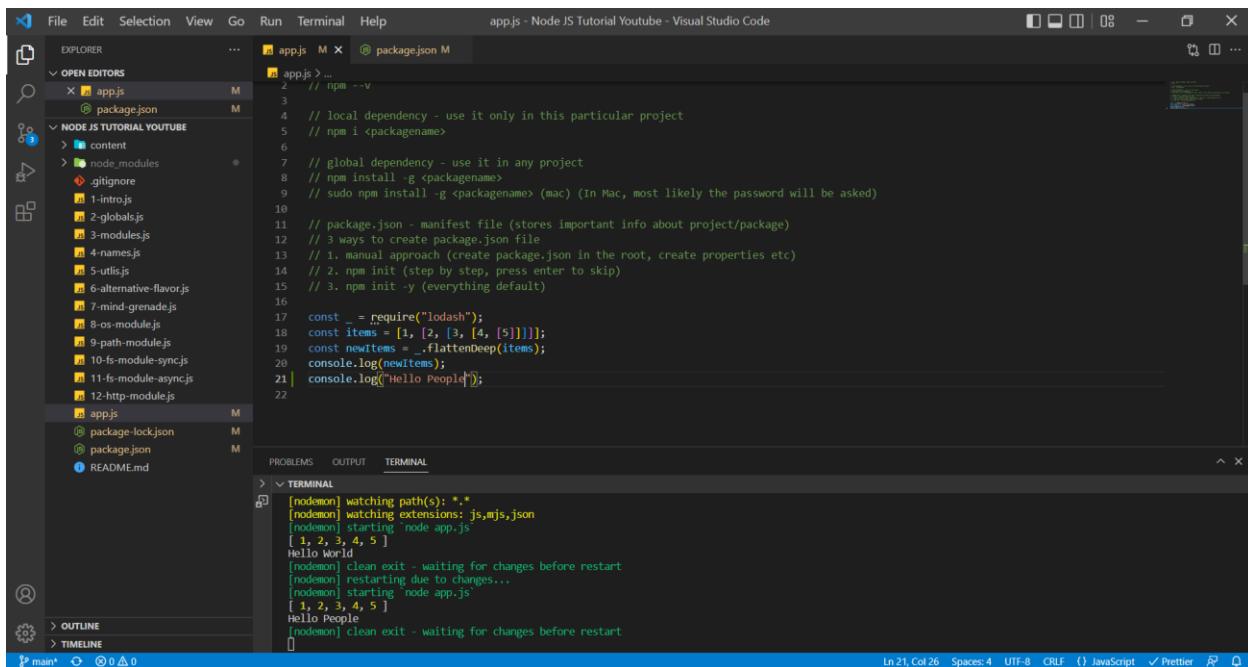
The screenshot shows the Visual Studio Code interface. The Explorer sidebar shows a project structure with files like `app.js`, `package.json`, and several `1-intro.js` through `12-http-module.js`. The `package.json` file is open in the editor, showing a `scripts` section with `"start": "node app.js"` and `"dev": "nodemon app.js"`. The terminal window shows the command `nodemon app.js` being run, and the output shows Nodemon starting the application and printing "Hello World".

```
1 {
2   "name": "nodejs-tutorial",
3   "version": "1.0.0",
4   "description": "",
5   "main": "1-intro.js",
6   "scripts": [
7     "start": "node app.js",
8     "dev": "nodemon app.js"
9   ],
10  "author": "sai krishna",
11  "license": "ISC",
12  "dependencies": {
13    "bootstrap": "^5.1.3",
14    "lodash": "^4.17.21"
15  },
16  "devDependencies": {
17    "nodemon": "^2.0.16"
18  }
19}
```

```
nodejs-tutorial@1.0.0 dev C:\Users\saikr\Documents\Node JS Tutorial Youtube
> nodemon app.js

[nodemon] 2.0.16
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): ***!
[nodemon] watching extensions: js,mjs,json
[nodemon] starting `node app.js`
[ 1, 2, 3, 4, 5 ]
Hello World
[nodemon] clean exit - waiting for changes before restart
```

With the Nodemon package, since it watches the file system and then any if change appears in our file system, it immediately restarts our application.



The screenshot shows the Visual Studio Code interface with the `package.json` file open. The `scripts` section now contains `"start": "node app.js"`. The `app.js` file has been modified to log "Hello People" instead of "Hello World". The terminal shows Nodemon restarting the application and printing "Hello People".

```
1 // npm --v
2 // local dependency - use it only in this particular project
3 // npm i <packagename>
4
5 // global dependency - use it in any project
6 // npm install -g <packagename>
7 // sudo npm install -g <packagename> (mac) (In Mac, most likely the password will be asked)
8
9 // package.json - manifest file (stores important info about project/package)
10 // 3 ways to create package.json file
11 // 1. manual approach (create package.json in the root, create properties etc)
12 // 2. npm init (step by step, press enter to skip)
13 // 3. npm init -y (everything default)
14
15 const _ = require("lodash");
16 const items = [1, [2, [3, [4, [5]]]]];
17 const newItems = _.flattenDeep(items);
18 console.log(newItems);
19
20 console.log("Hello People");
```

```
nodejs-tutorial@1.0.0 dev C:\Users\saikr\Documents\Node JS Tutorial Youtube
> nodemon app.js

[nodemon] watching path(s): ***!
[nodemon] watching extensions: js,mjs,json
[nodemon] starting `node app.js`
[ 1, 2, 3, 4, 5 ]
Hello World
[nodemon] clean exit - waiting for changes before restart
[nodemon] restarting due to changes...
[nodemon] starting `node app.js`
[ 1, 2, 3, 4, 5 ]
Hello People
[nodemon] clean exit - waiting for changes before restart
```

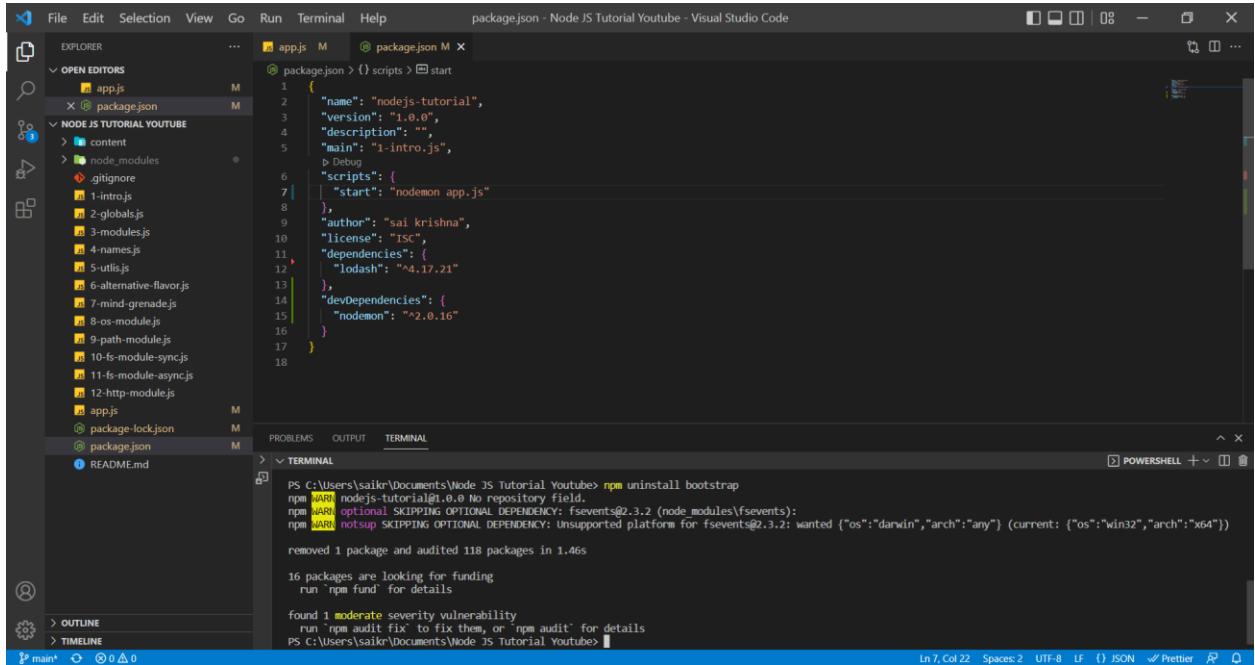
In the above example, we just changed the statement inside the `console.log` from "Hello World" to "Hello People" and Nodemon immediately recognized and restarted the application and executed the code.

**Use Control + C command to stop the Nodemon.**

**Uninstall a Package in package.json file**

There are two ways to uninstall a package.

1. Run command `npm uninstall <package-name>`



The screenshot shows the Visual Studio Code interface with the terminal tab active. The terminal window displays the command `npm uninstall bootstrap` being run. The output shows several warnings about optional dependencies and unsupported platforms, followed by a message indicating 1 package was removed and 118 packages were audited. The terminal also shows a moderate severity vulnerability found.

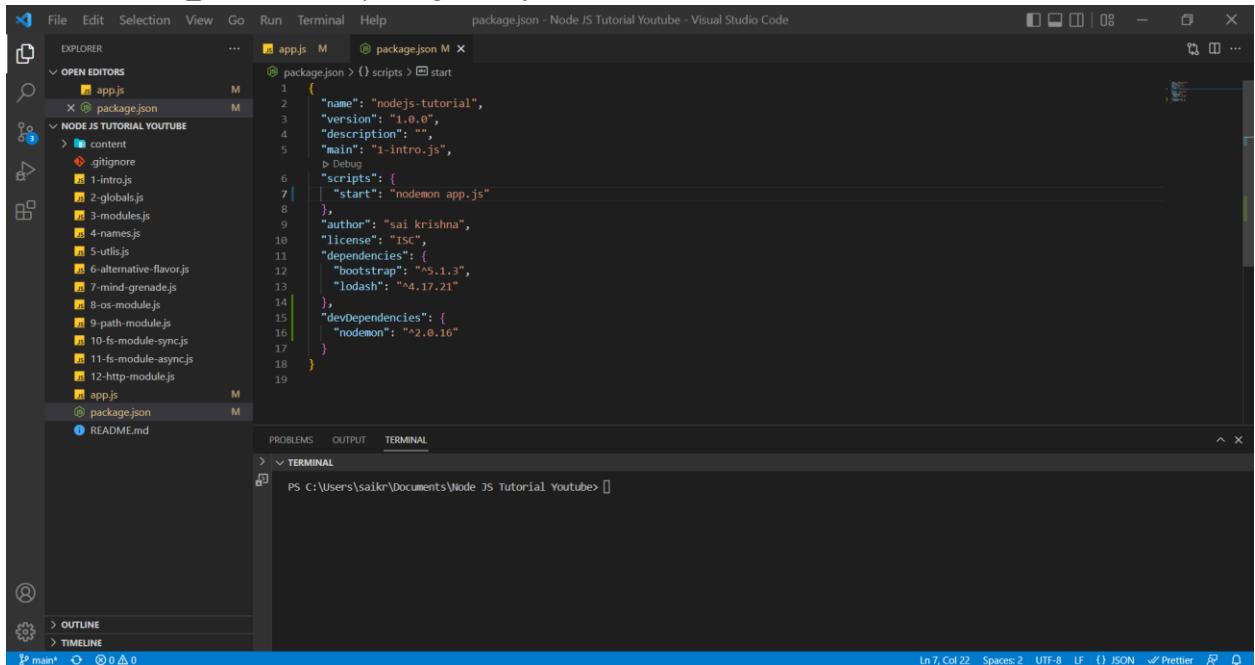
```
PS C:\Users\saikr\Documents\Node JS Tutorial Youtube> npm uninstall bootstrap
npm WARN nodejs-tutorial@1.0.0 No repository field.
npm WARN optional SKIPPING OPTIONAL DEPENDENCY: fsevents@2.3.2 (node_modules\fsevents):
npm WARN notsup NOTSUPPLIED OPTIONAL DEPENDENCY: Unsupported platform for fsevents@2.3.2: wanted {"os":"darwin","arch":"any"} (current: {"os":"win32","arch":"x64"})
removed 1 package and audited 118 packages in 1.46s

16 packages are looking for funding
  run npm fund for details

found 1 moderate severity vulnerability
  run npm audit fix to fix them, or npm audit for details
PS C:\Users\saikr\Documents\Node JS Tutorial Youtube>
```

2. Nuclear Approach is deleting node\_modules folder, package-lock.json file and removing the package name from package.json file and run command `npm install`

Removed node\_modules and package-lock.json file



The screenshot shows the Visual Studio Code interface with the terminal tab active. The terminal window is empty, indicating that the node\_modules folder and package-lock.json file have been removed.

```
PS C:\Users\saikr\Documents\Node JS Tutorial Youtube> []
```

Removing the package name (bootstrap) from package.json file

```

{
  "name": "nodejs-tutorial",
  "version": "1.0.0",
  "description": "",
  "main": "4-intro.js",
  "scripts": {
    "start": "nodemon app.js"
  },
  "author": "sai krishna",
  "license": "ISC",
  "dependencies": {
    "lodash": "^4.17.21"
  },
  "devDependencies": {
    "nodemon": "^2.0.16"
  }
}

```

PS C:\Users\saikr\Documents\Node JS Tutorial Youtube> [REDACTED]

Run command ***npm install*** (node\_modules will be installed and package-lock.json will be added to project)

```

{
  "name": "nodejs-tutorial",
  "version": "1.0.0",
  "description": "",
  "main": "4-intro.js",
  "scripts": {
    "start": "nodemon app.js"
  },
  "author": "sai krishna",
  "license": "ISC",
  "dependencies": {
    "lodash": "^4.17.21"
  },
  "devDependencies": {
    "nodemon": "^2.0.16"
  }
}

```

PS C:\Users\saikr\Documents\Node JS Tutorial Youtube> npm install

```

> nodemon@2.0.16 postinstall C:\Users\saikr\Documents\Node JS Tutorial Youtube\node_modules\nodemon
> node bin/postinstall || exit 0

npm notice created a lockfile as package-lock.json. You should commit this file.
npm warn optional SKIPPING OPTIONAL DEPENDENCY: fsevents@2.3.2 (node_modules\chokidar\node_modules\fsevents):
npm warn notsup SKIPPING OPTIONAL DEPENDENCY: Unsupported platform for fsevents@2.3.2: wanted {"os":"darwin","arch":"any"} (current: {"os":"win32","arch":"x64"})
npm warn nodejs-tutorial@1.0.0 No repository field.

added 117 packages from 56 contributors and audited 118 packages in 6.777s
16 packages are looking for funding

```

**Nuclear Approach is followed in Gatsby applications.**

## Global Install

In a scenario, assume we have 20 nodal applications, and we are constantly working on node applications, so every time we can't go and change our application name in the scripts object in package.json file. To make our work easier we make nodemon package global, where we can run command **nodemon <file-name>** to run an application and restart the application whenever there is a change in the file system.

To Install a package globally, we run the command in the following way

**npm install -g <file-name>**

Ex: **npm install -g nodemon**

Before nodemon was installed globally, we get the error because it can't recognize it globally.

The screenshot shows the Visual Studio Code interface. In the Explorer sidebar, there is a folder named 'NODE JS TUTORIAL YOUTUBE' containing several files like .gitignore, 1-intro.js, 2-globals.js, 3-modules.js, 4-names.js, 5-utilis.js, 6-alternative-flavor.js, 7-mind-grenade.js, 8-os-modules.js, 9-path-module.js, 10-fs-module-sync.js, 11-fs-module-async.js, 12-http-module.js, app.js, package-lock.json, package.json, and README.md. The package.json file is open in the main editor area, showing its contents:

```
1 {  
2   "name": "nodejs-tutorial",  
3   "version": "1.0.0",  
4   "description": "",  
5   "main": "1-intro.js",  
6   "scripts": {  
7     "start": "nodemon app.js"  
8   },  
9   "author": "sai krishna",  
10  "license": "ISC",  
11  "dependencies": {  
12    "lodash": "^4.17.21"  
13  },  
14  "devDependencies": {  
15    "nodemon": "^2.0.16"  
16  }  
17}  
18
```

Below the editor, the terminal window shows the output of running 'nodemon app.js':

```
PS C:\Users\saikr\Documents\Node JS Tutorial Youtube> nodemon app.js  
nodemon : The term 'nodemon' is not recognized as the name of a cmdlet, function, script file, or operable program. Check the spelling of the name, or if a path  
was included, verify that the path is correct and try again.  
At line:1 char:1  
+ nodemon app.js  
+ ~~~~~~  
+ CategoryInfo          : ObjectNotFound: (nodemon:String) [], CommandNotFoundException  
+ FullyQualifiedErrorId : CommandNotFoundException
```

After installing globally,

The screenshot shows the Visual Studio Code interface with the following details:

- File Explorer:** Shows the project structure with files like app.js, package.json, and README.md.
- Editor:** Displays the package.json file content:

```
1 {
  "name": "nodejs-tutorial",
  "version": "1.0.0",
  "description": "",
  "main": "1-intro.js",
  "scripts": {
    "start": "nodemon app.js"
  },
  "author": "sai krishna",
  "license": "ISC",
  "dependencies": {
    "lodash": "^4.17.21"
  },
  "devDependencies": {
    "nodemon": "^2.0.16"
  }
}
```

- Terminal:** Shows the command `npm install -g nodemon` being run, followed by the output indicating the package was installed successfully.

Run

command

*nodemon*

*app.js*

The screenshot shows the Visual Studio Code interface with the following details:

- File Explorer:** Shows the project structure with files like app.js, package.json, and README.md.
- Editor:** Displays the package.json file content.
- Terminal:** Shows the command `nodemon app.js` being run, followed by the output indicating the process is now listening for changes.

Before installing the nodemon package globally, it won't be recognized in the terminal but when we installed it globally it can be recognized.

Earlier we used to install frontend libraries like react and Gatsby globally but when **npx** came into light, the route has shifted by installing a package globally to use **npx** command.

Ex: `npx create-react-app my-app`

**npx** stands for Node Package Execute and official name is Package Runner. It is a feature that is introduced in NPM 5.2. The main idea is to be running the CLI tool without installing the packages globally.

### **package-lock.json**

If you remember some of the dependencies have their own dependencies like bootstrap which dependencies like jQuery and others etc. and those dependencies have versions as well.

Sometimes the dependencies of the packages may change as well by mistake when we handover code from developer to developer and it may introduce bugs into our application since there was change in package and dependencies versions.

Package-lock.json file contains all the versions of the packages and their dependencies as well.

```
"dependencies": {  
    "lodash": "^4.17.21"  
},
```

A package version has 3 values. This can be considered as contract between the person who is using the package and the person who developed.

The first number is the major change which means there are breaking changes. The second number is a minor one, which means backward compatible. If we change that to 18, we don't expect any breaking changes. The last one is patch for the bug fix. It is important to remember whenever we publish our own package.

### **Event Loops (A complex topic)**

The event loop is what allows Node JS to perform non-blocking I/O operations – even though JavaScript is single threaded – by **offloading** operations to the system kernel whenever possible.

## sync-js.js

The screenshot shows the Visual Studio Code interface with the title bar "sync-js.js - Node JS Tutorial Youtube - Visual Studio Code". The Explorer sidebar on the left lists files under "NODE JS TUTORIAL YOUTUBE" and "node\_modules". The "OPEN EDITORS" section shows two tabs: "sync-js.js" and "async-js.js". The "sync-js.js" tab is active, displaying the following code:

```
event-loop > sync-js.js > ...
1 // this piece of code is run on browser since we are using document which is DOM object
2 // Here JavaScript is Synchronous and Single Threaded
3 // It executes the next task only when it completes the second task (for loop) which is time consuming
4 console.log("first task");
5
6 console.time();
7 // we cannot offload for loops to the browser
8 for (let i = 0; i < 100000000; i++) {
9   const h3 = document.querySelector("h3");
10  h3.textContent = "Hey, Everyone is waiting on me";
11 }
12 console.timeEnd();
13
14 console.log("next task");
```

The terminal at the bottom shows the command "PS C:\Users\saikr\Documents\Node JS Tutorial Youtube\event-loop>" followed by the output of the script execution.

## async-js.js

The screenshot shows the Visual Studio Code interface with the title bar "async-js.js - Node JS Tutorial Youtube - Visual Studio Code". The Explorer sidebar on the left lists files under "NODE JS TUTORIAL YOUTUBE" and "node\_modules". The "OPEN EDITORS" section shows two tabs: "sync-js.js" and "async-js.js". The "async-js.js" tab is active, displaying the following code:

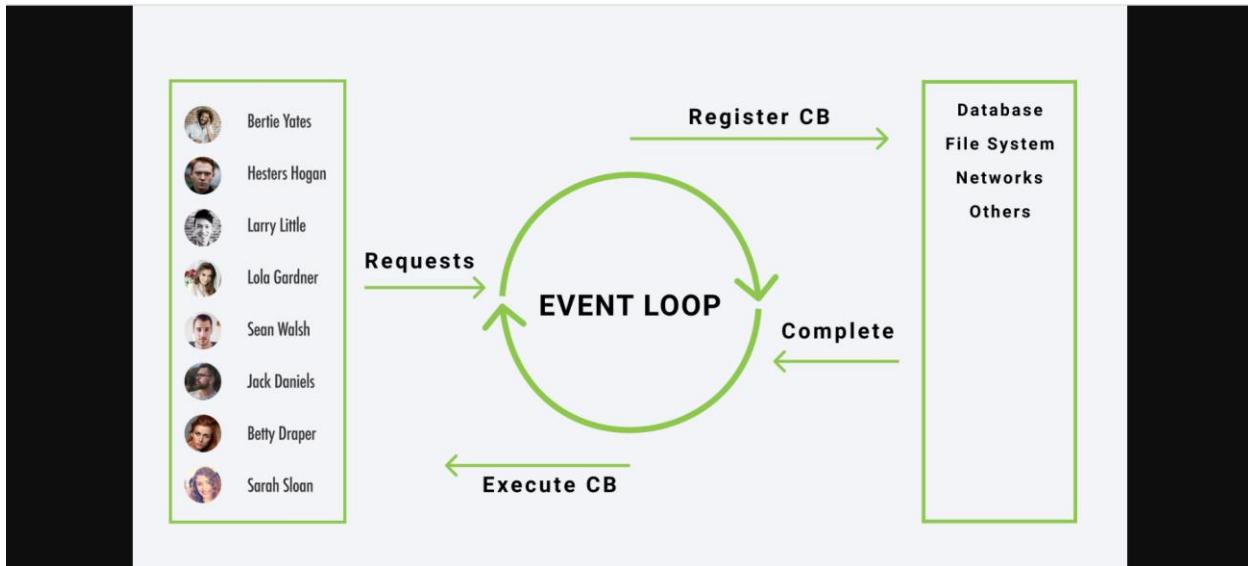
```
event-loop > sync-js.js > ...
1 // In browsers we use the API call, so we offload the task and execute the callback function when the task is completed
2 // using fetch is good example for the statement
3 // but still we can get an idea by using the below task
4 console.log("first task");
5
6 // even we set the time to be 0 seconds, callback will be executed only when the next task is completed
7 setTimeout(() => {
8   console.log("second task");
9 }, 0);
10
11 console.log("next task");
12
```

The terminal at the bottom shows the command "PS C:\Users\saikr\Documents\Node JS Tutorial Youtube\event-loop> node async-js" followed by the output of the script execution.

By saying we cannot offload for loop, we can still write blocking code in JavaScript but the browser does provide some nice API's where we can offload the time consuming tasks.

## Node JS Event Loop

Assume a scenario where our application is popular and there are 8 users who are requesting information from application and as the requests are coming in the event loop is responsible for avoiding this type of scenario.



In the above scenario, assume Larry Little requesting something from our application which is time consuming, so in this case event loop registers the callback function basically it registers what needs to be done when the task is complete because if event loop wouldn't do that then we would have the above scenario where the request are coming in and because Larry is requesting something that take a long time then rest of the users have to wait.

It's just a fact that we are wasting our time on waiting for the operation to be done and then only we can serve other users but what event loop does is it registers the callback and only when the operation is complete it executes the callback function.

Here the requests are coming in, let's say the operation is complete we first register the callback and operation is complete and instead of executing the callback right away it effectively puts away at the end of the line and then when there is no immediate code to run then we execute the callback.

***With the help of Event Loop, we can offload some time-consuming operations and effectively just keep our other requests executed instead of wasting time for the time-consuming operation to complete.***

## Event Loop Example Codes

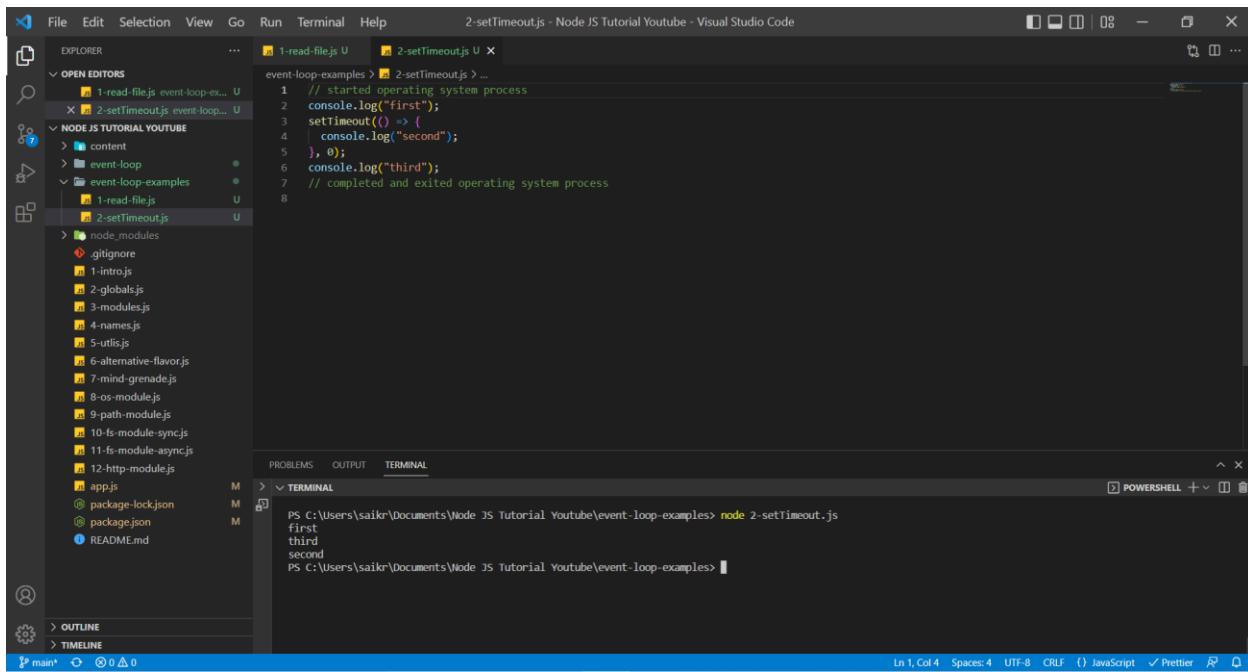
Here the Event Loop registers the callback function. Event Loop will offload this task (in this case, it will offload to file system). Event Loop will execute the next tasks and in the meanwhile if the readFile operation is complete ( it may provide error or data ) Event Loop puts that operation at end of the process/list and executes the remaining tasks. After all the tasks are executed and when there are no immediate lines to be executed then Event Loop executes the registered callback function. That is the reason you will see **result** and **completed first task** statement at the bottom of the screen.

### 1-read-file.js

The screenshot shows the Visual Studio Code interface with the following details:

- File Explorer:** Shows the project structure under "NODE JS TUTORIAL YOUTUBE". The "event-loop-examples" folder contains several files: 1-read-file.js, 2-globals.js, 3-modules.js, 4-names.js, 5-util.js, 6-alternative-flavor.js, 7-mind-grenade.js, 8-os-module.js, 9-path-module.js, 10-fs-module-sync.js, 11-fs-module-async.js, 12-http-module.js, app.js, package-lock.json, package.json, and README.md.
- Editor:** The "1-read-file.js" file is open. The code reads a file named "first.txt" and logs its content to the console. It includes comments explaining the event loop behavior: starting tasks, offloading to the file system, and executing remaining tasks.
- Terminal:** The terminal window shows the command "node 1-read-file.js" being run, followed by the output of the program. The output shows the file content being read and then the message "Completed the first task".

## 2-setTimeout.js



```
// started operating system process
console.log("first");
setTimeout(() => {
  console.log("second");
}, 0);
console.log("third");
// completed and exited operating system process
```

PROBLEMS    OUTPUT    TERMINAL

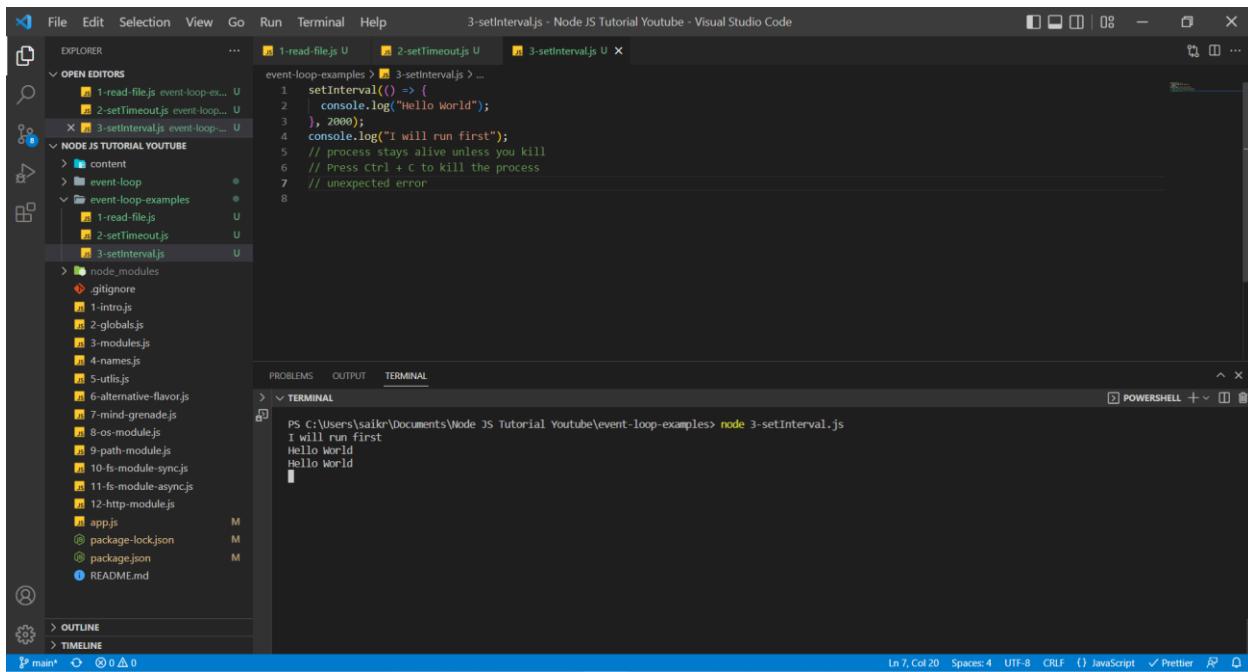
PS C:\Users\saikr\Documents\Node JS Tutorial Youtube\event-loop-examples> node 2-setTimeout.js

first  
third  
second

PS C:\Users\saikr\Documents\Node JS Tutorial Youtube\event-loop-examples>

Even though `setTimeout` has 0 seconds (it is asynchronous code), the Event Loop will offload the task (in this case it will be offloaded to operating system) and execute the immediate tasks (synchronous code) and then there is no immediate task to execute then Event loop will execute the offloaded task (asynchronous code).

### 3-setInterval.js



```
event-loop-examples > 3-setInterval.js > ...
1 setinterval(() => {
2   | console.log("Hello World");
3 }, 2000);
4 console.log("I will run first");
5 // process stays alive unless you kill
6 // Press Ctrl + C to kill the process
7 // unexpected error
8
```

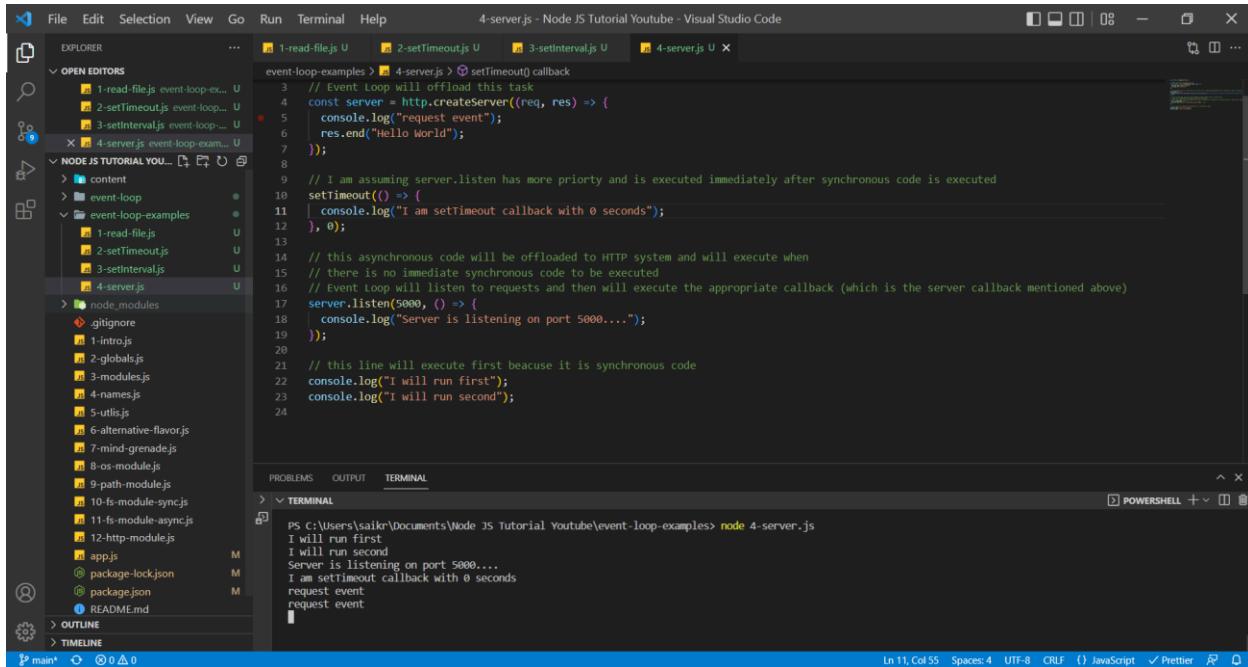
PROBLEMS    OUTPUT    TERMINAL

```
PS C:\Users\saikr\Documents\Node JS Tutorial Youtube\event-loop-examples> node 3-setInterval.js
I will run first
Hello World
Hello World
```

Ln 7, Col 20   Spaces: 4   UTF-8   CR LF   { } JavaScript   ✓ Prettier

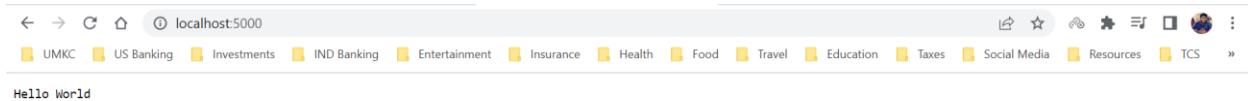
setInterval runs in intervals. So that's why we see the console statement after 2 every seconds. It will be offloaded to Event Loop and then executed when there is no immediate synchronous code.

## 4-server.js



```
event-loop-examples > 4-server.js > setTimeout() callback
3 // Event Loop will offload this task
4 const server = http.createServer((req, res) => {
5   console.log("request event");
6   res.end("Hello World");
7 });
8
9 // I am assuming server.listen has more priority and is executed immediately after synchronous code is executed
10 setTimeout(() => {
11   console.log("I am setTimeout callback with 0 seconds");
12 }, 0);
13
14 // this asynchronous code will be offloaded to HTTP system and will execute when
15 // there is no immediate synchronous code to be executed
16 // Event Loop will listen to requests and then will execute the appropriate callback (which is the server callback mentioned above)
17 server.listen(5000, () => {
18   console.log("Server is listening on port 5000....");
19 });
20
21 // this line will execute first because it is synchronous code
22 console.log("I will run first");
23 console.log("I will run second");
24
```

## Request from Webpage

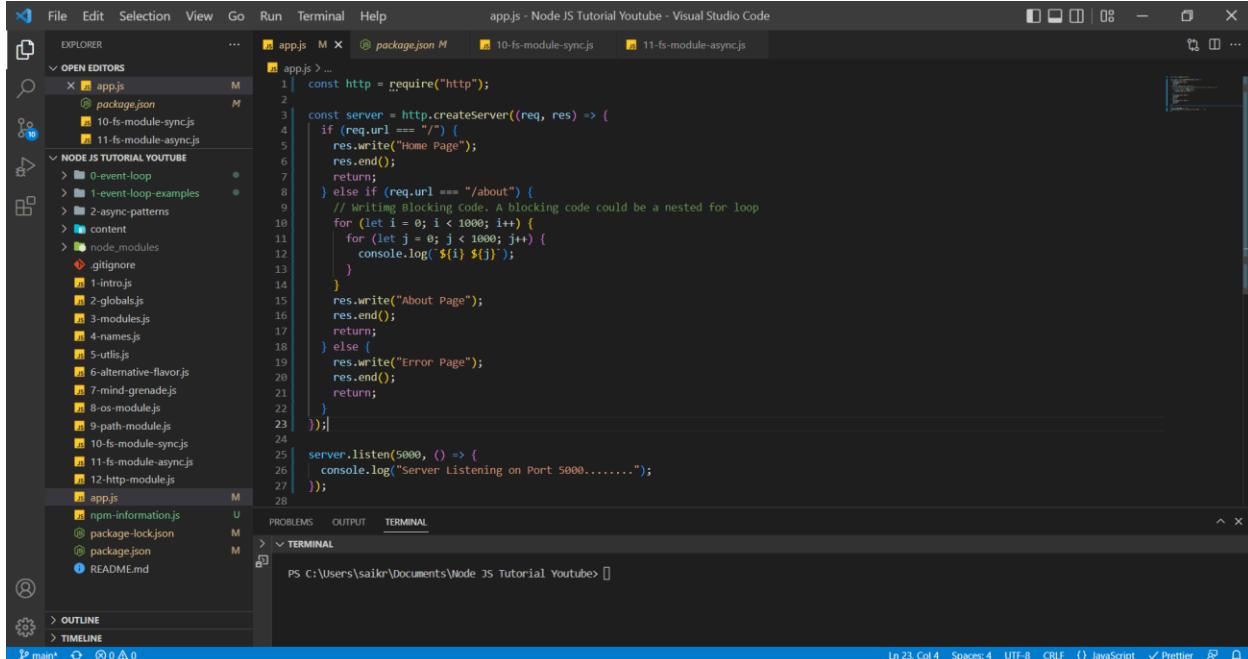


Here, we are starting the server and then listening to the server in port 5000 and whenever we listen a request from the server, we execute the appropriate callback function and return the response.

server.listen() process stays alive because listen is an asynchronous function and the moment, we set it up, Event Loop is just waiting for those requests to come in and once they come in Event Loop runs the appropriate callback function and sends the response back to the server. This process will end when we terminate or close the terminal.

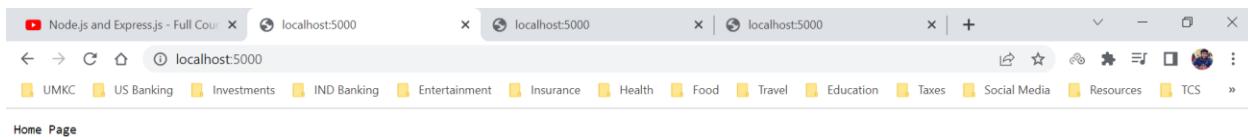
## Async Patterns in Node JS

As we already discussed callback hell in File System Module. Callback Hell is nothing but implementing one callback function inside another callback function and going on. Callback Hell will mess up the code and hence we use different approaches to solve the code blocking in JavaScript.



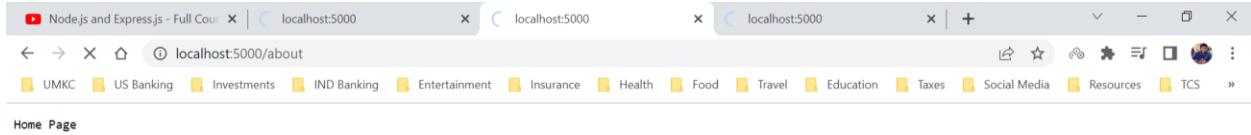
```
const http = require("http");
const server = http.createServer((req, res) => {
  if (req.url === "/") {
    res.write("Home Page");
    res.end();
    return;
  } else if (req.url === "/about") {
    // Writing Blocking code. A blocking code could be a nested for loop
    for (let i = 0; i < 1000; i++) {
      for (let j = 0; j < 1000; j++) {
        console.log(`${i} ${j}`);
      }
    }
    res.write("About Page");
    res.end();
    return;
  } else {
    res.write("Error Page");
    res.end();
    return;
  }
});
server.listen(5000, () => {
  console.log("Server Listening on Port 5000....");
});
```

In the above code, a server is created and listening to the port at 5000 and responds to the request for three URLs. We can login from multiple tabs and hit the server with a request and we will be served. This works fine until there is no blocking code, but once we have blocking code in any of the request block then we assume that only the user requesting URL but not. In real time, all the users request the server will be blocked since server is servicing the most time-consuming request which is in a synchronous code format.

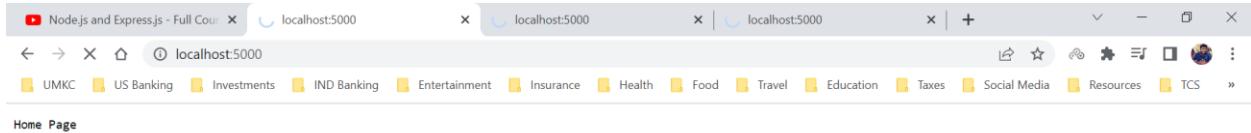


We have 3 different tabs or users requesting the server a response. Server is proving them with a response. In the code we have a blocking piece code for **localhost:5000/about** URL and when a user requests a response from that URL (`localhost:5000/about`) the requests from other users will also be blocked and response will be provided only that time-consuming request is completed.

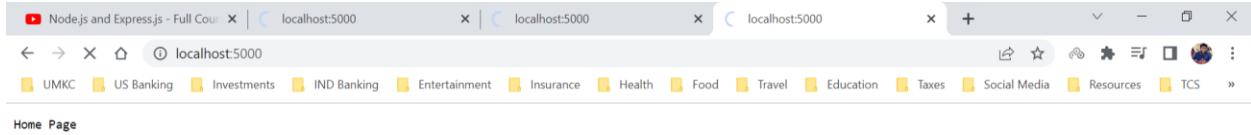
From User 2/ Tab 2 we are requesting About Page (which contains the blocking code)



After that, we requested Home Page from User 1/ Tab 1 and our page is still loading even though Home Page block doesn't contain any blocking code.



In the same way, User 3/ Tab 3 request Home Page but still page will be in loading state.



User 1 and User 3 request will be resolved only User 2 request is resolved.

***This is the representation why prefer the asynchronous approach. We should always strive to set our code asynchronously.***

## Async Pattern – Promise Setup

The screenshot shows the Visual Studio Code interface. The Explorer sidebar on the left lists files in the 'app.js - Node JS Tutorial Youtube' folder, including 'app.js', 'node\_modules', and several '1-\*' and '2-\*' files. The 'app.js' file is open in the editor, showing a simple asynchronous function that reads 'first.txt'. The terminal at the bottom shows the execution of 'node app.js' followed by its output: 'Hello this is first text file'. The status bar at the bottom right indicates the file is in 'JavaScript' mode.

```
const (readFile) = require('fs');

readFile("./content/first.txt", "utf8", (err, data) => {
  if (err) {
    return;
  } else {
    console.log(data);
  }
});
```

```
[nodemon] 2.0.16
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *
[nodemon] watching extensions: js,mjs,json
[nodemon] starting `node app.js`
Hello this is first text file
[nodemon] clean exit - waiting for changes before restart
```

The above code works well but the real problem starts when we want to read multiple files and write data into one file.

The better solution would be turning that into a promise.

This is how an asynchronous function inside a promise looks like. Promise is divided in 3 stages which are

### Pending, Reject and Resolve

The screenshot shows the Visual Studio Code interface. The Explorer sidebar lists files including '3-promise-pattern.js'. The '3-promise-pattern.js' file is open in the editor, showing a Promise-based function 'getText'. The terminal at the bottom shows the execution of 'node 3-promise-pattern.js' followed by its output: 'Hello this is second text file'. The status bar at the bottom right indicates the file is in 'powershell' mode.

```
const getText = (path) => {
  return new Promise((resolve, reject) => {
    readFile(path, "utf8", (err, data) => {
      if (err) {
        reject(err);
      } else {
        resolve(data);
      }
    });
  });
};

getText("./content/second.txt")
  .then(result => console.log(result))
  .catch(err => console.log(err));

getText("./content/first.txt")
  .then(result => console.log(result))
  .catch(err => console.log(err));
```

```
PS C:\Users\salikr\Documents\Node JS Tutorial Youtube\2-async-patterns> node 3-promise-pattern
[Error: ENOENT: no such file or directory, open 'C:\Users\salikr\Documents\Node JS Tutorial Youtube\2-async-patterns\content\first.txt' ]
  errno: -4058,
  code: 'ENOENT',
  syscall: 'open',
  path: 'C:\Users\salikr\Documents\Node JS Tutorial Youtube\2-async-patterns\content\first.txt'
)
Hello this is second text file
PS C:\Users\salikr\Documents\Node JS Tutorial Youtube\2-async-patterns>
```

If Promise is fulfilled, it is Resolved. If it isn't fulfilled or failed, then it is considered as Reject. If it is in either of those two stages, then it would be in Pending State.

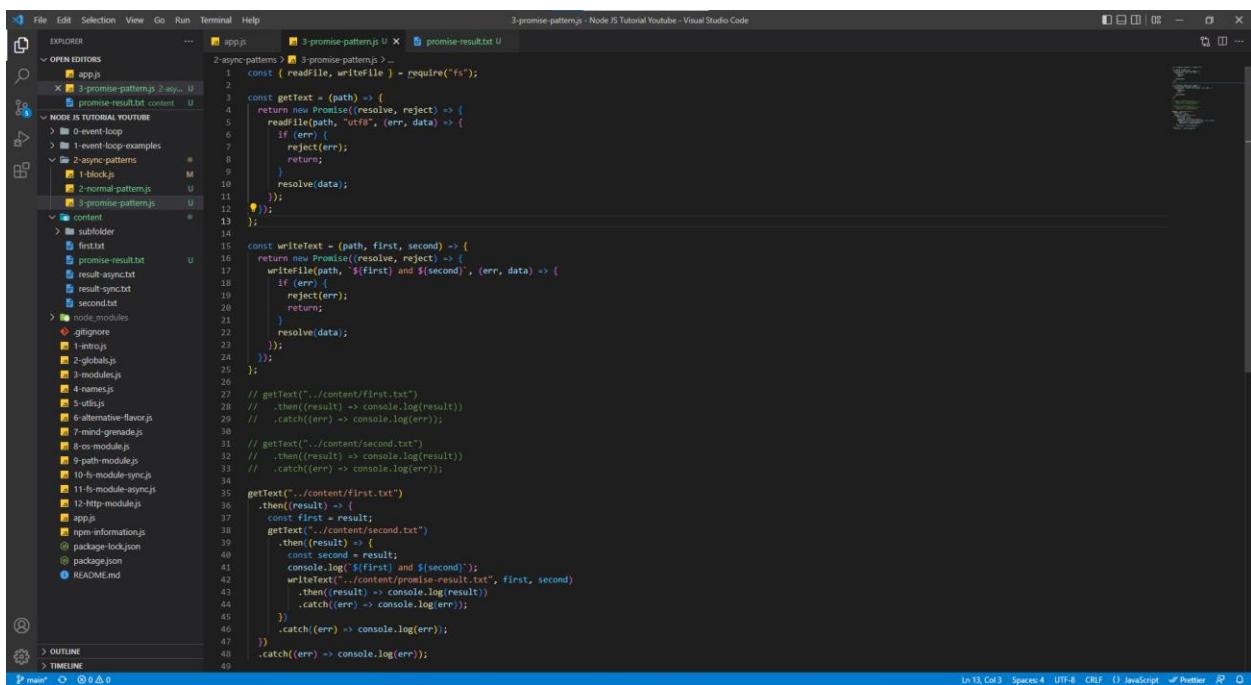
If a Promise is resolved it will enter then block, if it rejected it will enter catch block. The above screenshot depicts Resolved and Rejected states of Promise.

***The above code is still not cleaner and that is why we follow different approach of setting up Async and Await.***

If we want to do 2 file reads and 1 write still it would be messier because we need to chain promises one inside another, hence we use the approach Async and Await.

Chaining of Promises can be found below,

### 3-promise-pattern.js



```
const { readFile, writeFile } = require("fs");

const getText = (path) => {
  return new Promise((resolve, reject) => {
    readFile(path, "utf8", (err, data) => {
      if (err) {
        reject(err);
        return;
      }
      resolve(data);
    });
  });
};

const writeText = (path, first, second) => {
  return new Promise((resolve, reject) => {
    writeFile(path, `${first} and ${second}`, (err, data) => {
      if (err) {
        reject(err);
        return;
      }
      resolve(data);
    });
  });
};

// getText("../content/first.txt")
// .then(result) => console.log(result)
// .catch(err) => console.log(err);

// getText("../content/second.txt")
// .then(result) => console.log(result)
// .catch(err) => console.log(err);

// getText("../content/first.txt")
// .then(result) => {
//   const first = result;
//   getText("../content/second.txt")
//     .then(result) => {
//       const second = result;
//       console.log(`${first} and ${second}`);
//       writeText("../content/promise-result.txt", first, second)
//         .then(result) => console.log(result)
//         .catch(err) => console.log(err);
//     }
//   .catch(err) => console.log(err);
// }
// .catch(err) => console.log(err);

// promise-result.txt
```

and the result can be seen in terminal and promise-result.txt file

terminal

The screenshot shows the Visual Studio Code interface. The Explorer sidebar on the left lists files and folders related to a 'Node JS Tutorial YouTube' project. In the center, the code editor displays '3-promise-pattern.js'. The terminal at the bottom shows the following output:

```
PS C:\Users\saikir\Documents\Node JS Tutorial YouTube\2-async-patterns> node 3-promise-pattern
Hello this is first text file and Hello this is second text file
undefined
```

promise-result.txt

The screenshot shows the Visual Studio Code interface. The Explorer sidebar on the left lists files and folders related to a 'Node JS Tutorial YouTube' project. In the center, the code editor displays 'promise-result.txt'. The terminal at the bottom shows the following output:

```
PS C:\Users\saikir\Documents\Node JS Tutorial YouTube\2-async-patterns> node 3-promise-pattern
Hello this is first text file and Hello this is second text file
undefined
```