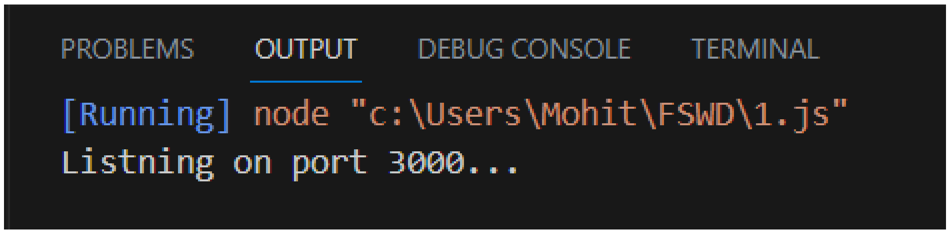
Task-1

Aim:
Setting up a basic HTTP server: Create a Node.js application that listens for incoming HTTP requests and responds with a simple message.

Source Code:

```
const http = require("http");
const httpserver = http.createServer(function(req,res){
    if(req.method == 'POST')
    {
        res.end("This is post request");
    }
});
httpserver.listen(3000,()=>{
    console.log("Listning on port 3000...");
})
```

Output:



```
[Running] node "c:\Users\Mohit\FSWD\1.js"
Listning on port 3000...
```

Theoretical Background:

1) const http = require("http");
   This line imports the built-in Node.js module http, which provides functionality for creating HTTP servers and making HTTP requests.

2) const httpserver = http.createServer(function(req,res){
   This line creates an HTTP server using the createServer method provided by the http module. It takes a callback function as an argument, which will be called whenever a request is made to the server. The callback function takes two arguments: req (the request object) and res (the response object).

3)
```
if(req.method == 'POST')
{
        res.end("This is post request");
}
```
   Within the callback function, this block of code checks if the request method is POST using req.method. If it is a POST request, the server sends back the response with the message "This is post request" using res.end(). The res.end() method is used to end the response and send the specified data back to the client.

4)
```
httpserver.listen(3000,()=>{
console.log("Listning on port 3000...");
})
```

   This line starts the server listening on port 3000 using the listen method. It takes two arguments: the port number to listen on (3000 in this case), and a callback function that will be executed once the server starts listening. In this case, it simply logs a message to the console indicating that the server is listening on port 3000.

   So, when you run this code, it creates an HTTP server that listens for requests on port 3000. If a POST request is made to the server, it responds with the message "This is post request".

Task-2

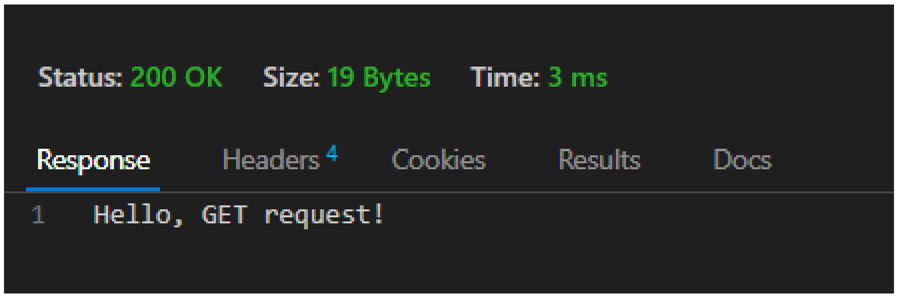
Aim :
Experiment with Various HTTP Methods,Content Types and Status Code

Source Code:

HTTP Methods:

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

const server = http.createServer((req, res) => {
  // GET request handler
  if (req.method === 'GET') {
    res.writeHead(200, { 'Content-Type': 'text/plain' });
    res.end('Hello, GET request!');
  }
  // POST request handler
  else if (req.method === 'POST') {
    res.writeHead(200, { 'Content-Type': 'text/plain' });
    res.end('Hello, POST request!');
  }
  // PUT request handler
  else if (req.method === 'PUT') {
    res.writeHead(200, { 'Content-Type': 'text/plain' });
    res.end('Hello, PUT request!');
  }
  // DELETE request handler
  else if (req.method === 'DELETE') {
    res.writeHead(200, { 'Content-Type': 'text/plain' });
    res.end('Hello, DELETE request!');
  }

 else if (req.method === 'PATCH') {
    res.writeHead(200, { 'Content-Type': 'text/plain' });
    res.end('Hello, PATCH request!');
  }
  // HEAD request handler
  else if (req.method === 'HEAD') {
    res.writeHead(200, { 'Content-Type': 'text/plain' });
    res.end('Hello, HEAD request!');
  }

  // OPTIONS request handler
  else if (req.method === 'OPTIONS') {
    res.writeHead(200, { 'Content-Type': 'text/plain' });
    res.end('Hello, OPTIONS request!');
  }

  // PROPFIND request handler
  else if (req.method === 'PROPFIND') {
    res.writeHead(200, { 'Content-Type': 'text/plain' });
    res.end('Hello, PROPFIND request!');
  }

  // Invalid request method
  else {
    res.writeHead(400, { 'Content-Type': 'text/plain' });
    res.end('Invalid request method');
  }
});

server.listen(3000, () => {
  console.log('Server is running on port 3000');
});
```

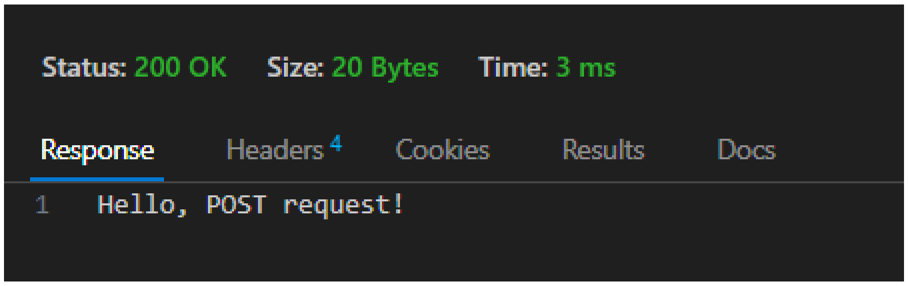
Output:
HTTP Methods:
GET:



Status: 200 OK   Size: 19 Bytes   Time: 3 ms

```
Hello, GET request!
```

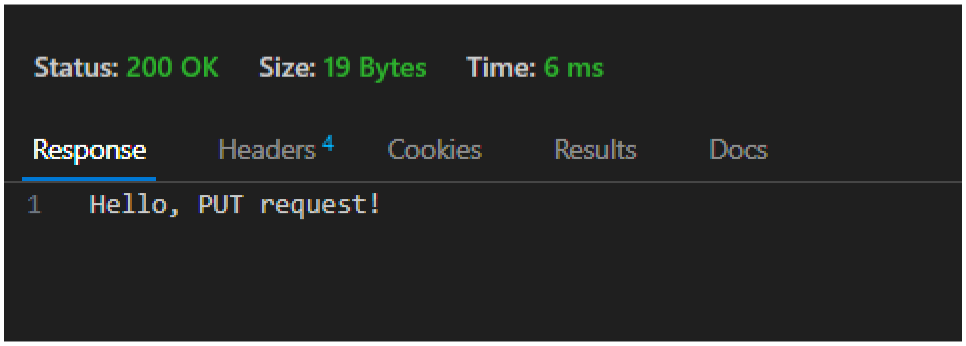
POST:



```
Status: 200 OK    Size: 20 Bytes    Time: 3 ms
```

```
Hello, POST request!
```

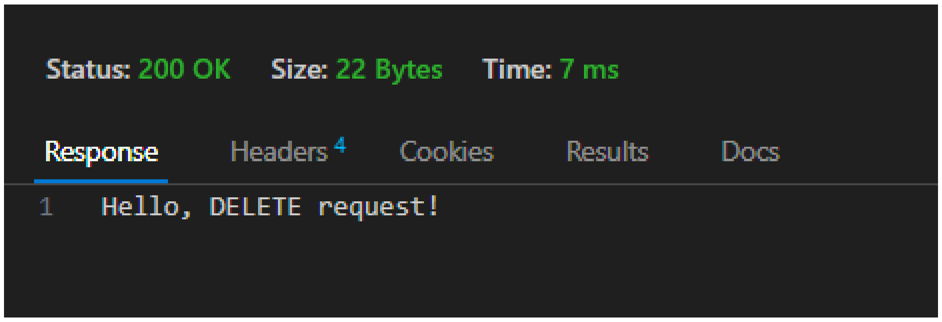
PUT:



```
Status: 200 OK    Size: 19 Bytes    Time: 6 ms
```

```
Hello, PUT request!
```

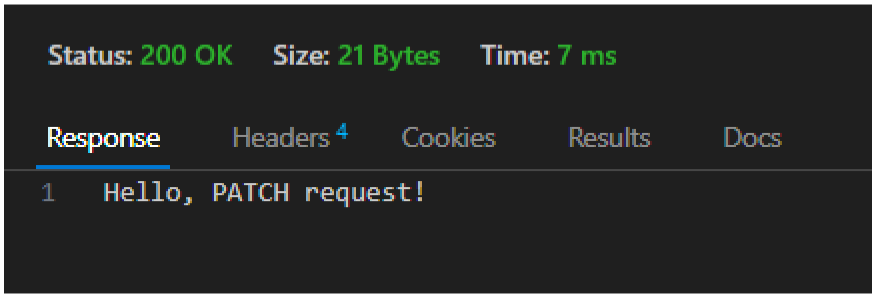
DELETE:



```
Status: 200 OK    Size: 22 Bytes    Time: 7 ms
```

```
Hello, DELETE request!
```

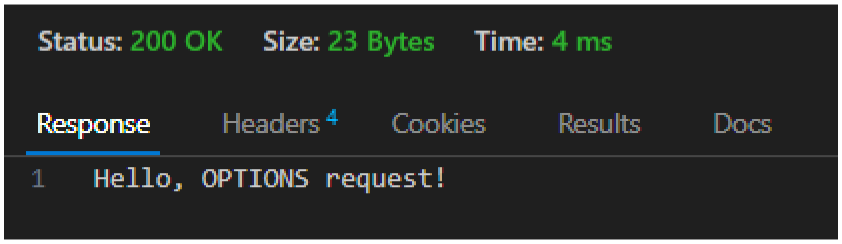
PATCH:



```
Status: 200 OK    Size: 21 Bytes    Time: 7 ms
```

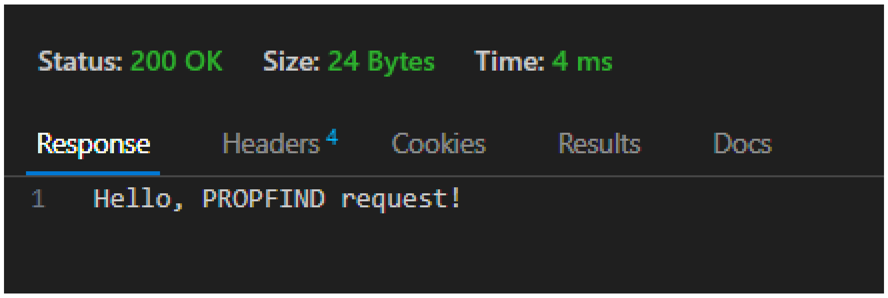
```
Hello, PATCH request!
```

OPTIONS:



```
Hello, OPTIONS request!
```

PROPFIND:



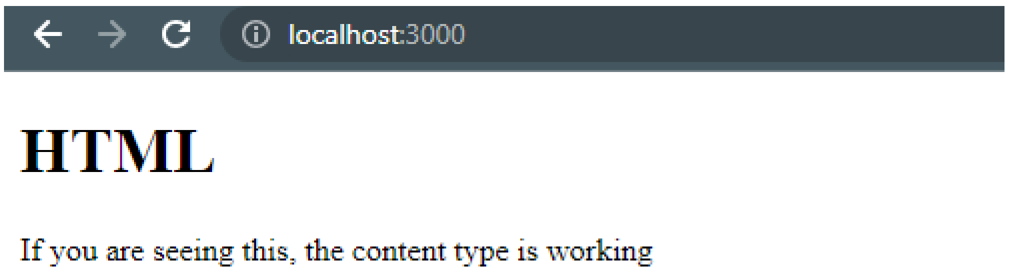
```
Hello, PROPFIND request!
```

Source Code:

Content Type:

```
const http = require('http')
const fs = require('fs')

http.createServer((req,res)=>{
   const readStream = fs.createReadStream('./static/index.htm')
// Assume we have a static folder having 3 static files 1)example.json 2)example.png
3)index.htm
   res.writeHead(200,{'content-type':'text/html'})
   readStream.pipe(res)
}).listen(3000);
```

Output:



HTML

If you are seeing this, the content type is working

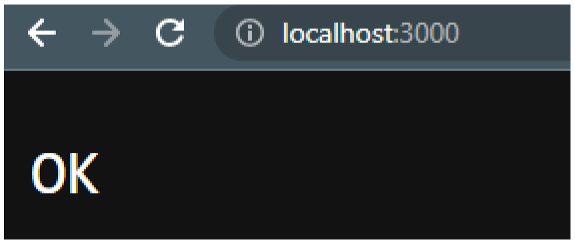
Source Code:

Status Code:

```
if (req.url === '/') {
    res.statusCode = 200;
    res.end('OK');
  } else if (req.url === '/not-found') {
    res.statusCode = 404;
    res.end('Not Found');
  } else if (req.url === '/server-error') {
    res.statusCode = 500;
    res.end('Internal Server Error');
  } else {
    res.statusCode = 400;
    res.end('Bad Request');
  }
});

server.listen(3000, () => {
  console.log('Server is running on port 3000');
});
```

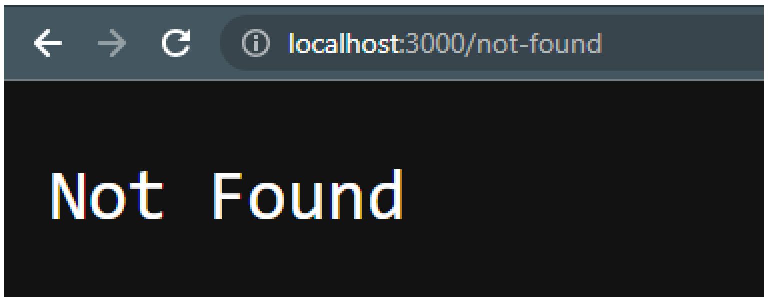
Output:

200


localhost:3000

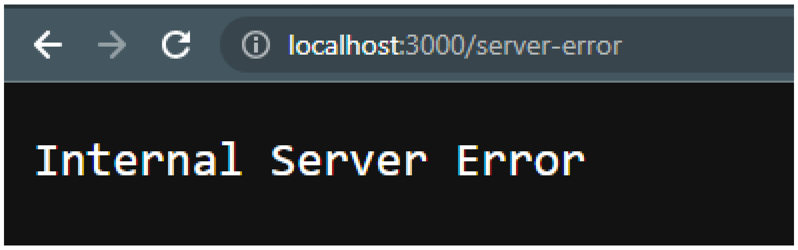
OK

404


localhost:3000/not-found

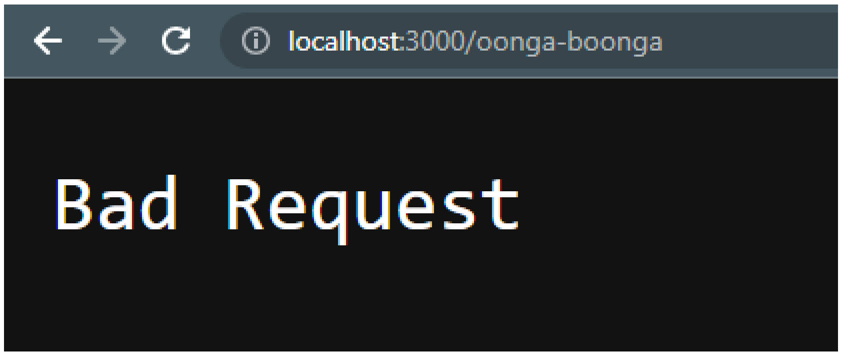
Not Found

500



```
Internal Server Error
```

400



```
Bad Request
```

Theoretical Background:

HTTP Methods:
The common HTTP methods are GET, POST, PUT, DELETE, and more. Each method has a different purpose and usage:

- GET is used to retrieve information from a server.
- POST is used to send data to the server to create new resources.
- PUT is used to send data to the server to update or replace existing resources.
- DELETE is used to request the removal of a resource from the server.

Content Types:
HTTP requests and responses often include a Content-Type header, indicating the type of data being sent or received. Common content types include :

- application/json
- application/xml
- text/html
- multipart/form-data, etc.

You can experiment with different content types by setting the Content-Type header accordingly in your requests.

Status Codes:
HTTP status codes provide information about the outcome of an HTTP request. Each status code represents a specific situation or condition. For example:

- Informational responses (100 – 199)
- Successful responses (200 – 299)
- Redirection messages (300 – 399)
- Client error responses (400 – 499)
- Server error responses (500 – 599)

Additional Information:

The writeHead function takes two arguments: the status code and an object containing the response headers.

res.writeHead(statusCode, headersObject);

statusCode is a numeric value representing the HTTP status code to be sent in the response. It indicates the outcome of the request, such as 200 for a successful request, 404 for a not found error, etc.

headersObject is an optional object that specifies the response headers. These headers provide additional information about the response, such as the content type, caching directives, cookies, and more.

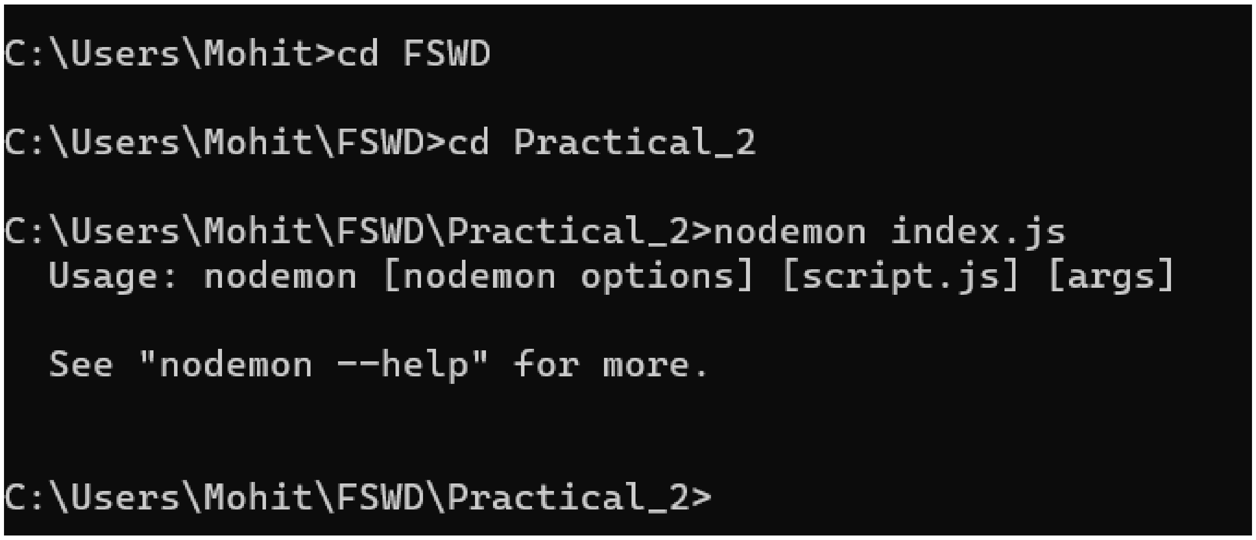
Task-3

Aim :
Test it using browser ,CLI and REST Client.

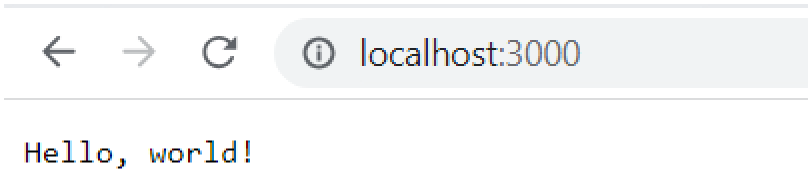
Source Code:

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

const server = http.createServer((req, res) => {
  res.writeHead(200, { 'Content-Type': 'text/plain' });
  res.end('Hello, world!');
});

server.listen(3000, () => {
  console.log('Server running on http://localhost:3000');
});
```

Output:



```
C:\Users\Mohit>cd FSWD

C:\Users\Mohit\FSWD>cd Practical_2

C:\Users\Mohit\FSWD\Practical_2>nodemon index.js
  Usage: nodemon [nodemon options] [script.js] [args]

  See "nodemon --help" for more.


C:\Users\Mohit\FSWD\Practical_2>
```



```
Hello, world!
```

Theoretical Background:

Testing the script: You can test your Node.js script in multiple ways.

Browser: If your Node.js script is designed to work in a browser environment, you can create an HTML file that includes the script and open it in a web browser to see the script's output or functionality.

CLI: Open a command prompt or terminal window, navigate to your project directory, and run the following command:

node index.js

This command will execute the "index.js" script using the Node.js runtime and display any output or errors in the command prompt or terminal window.

REST Client: If your Node.js script exposes an API or a web service, you can test it using a REST client tool such as Postman or cURL. These tools allow you to send HTTP requests to your script's API endpoints and inspect the responses.

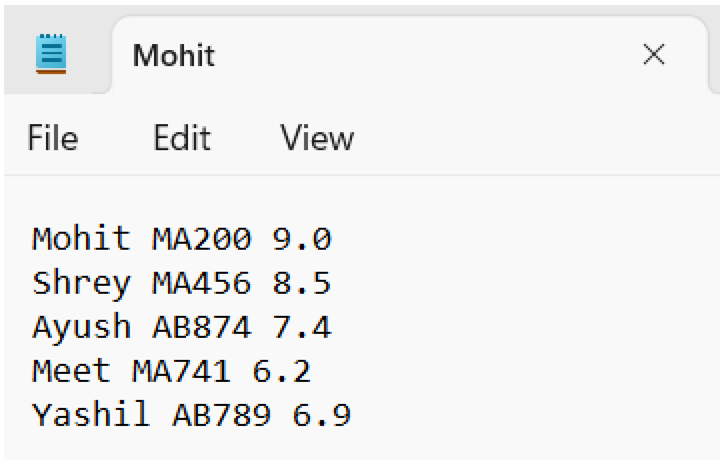
Task-4

Aim :
Read File student-data.txt file and find all students whose name contains ‘MA’ and CGPA > 7.

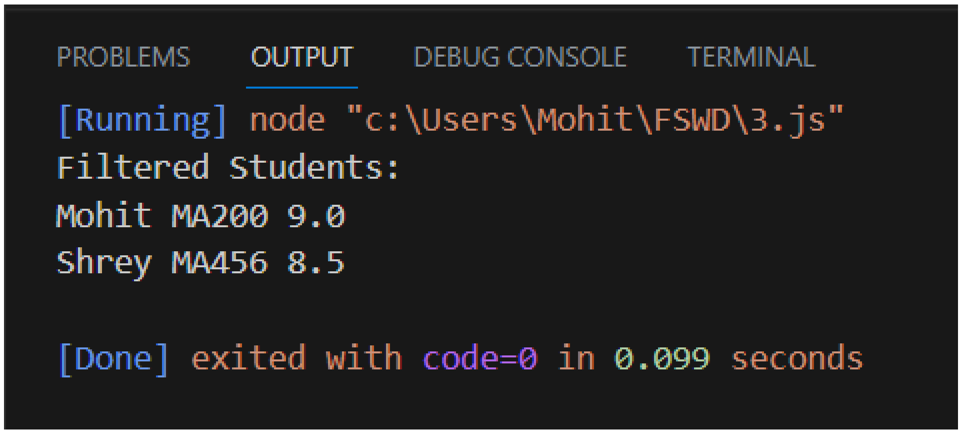
Source Code:

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

fs.readFile('mohit.txt', 'utf8', (err, data) => {
  if (err) {
    console.error('Error reading file:', err);
    return;
  }

  const lines = data.split('\n');

  console.log('Filtered Students:');
  lines.forEach((line) => {
    const [name, id, cgpa] = line.split(' ');
    if (id.includes('MA') && parseFloat(cgpa) > 7) {
      console.log(line);
    }
  });
});
```

Output:



```
Mohit MA200 9.0
Shrey MA456 8.5
Ayush AB874 7.4
Meet MA741 6.2
Yashil AB789 6.9
```



```
[Running] node "c:\Users\Mohit\FSWD\3.js"
Filtered Students:
Mohit MA200 9.0
Shrey MA456 8.5

[Done] exited with code=0 in 0.099 seconds
```

Theoretical Background:

1) const fs = require('fs');
This line imports the built-in Node.js module fs (file system module), which provides methods for interacting with the file system. It allows you to read and write files, among other file-related operations.

2)const lines = data.split('\n');
Assuming no error occurred, this code splits the content of the file into an array of lines using the split method. Each line in the file is separated by a new line character ('\n').

```
3)  console.log('Filtered Students:');
  lines.forEach((line) => {
    const [name, id, cgpa] = line.split(' ');
    if (id.includes('MA') && parseFloat(cgpa) > 7) {
      console.log(line);
    }
  });
});
```

After splitting the content into lines, this code begins iterating over each line using the forEach method of the lines array. For each line, it splits the line into separate components (name, ID, and CGPA) using the split method, with the space character (' ') as the separator.

Then, it checks if the ID component (id) includes the string 'MA' and if the CGPA component (cgpa) parsed as a floating-point number is greater than 7. If both conditions are true, it prints the entire line using console.log(line).

This code filters and prints the lines that contain the substring 'MA' in the ID component and have a CGPA greater than 7.

Please note that the explanation assumes the file 'student-data.txt' exists and contains data in the specified format.

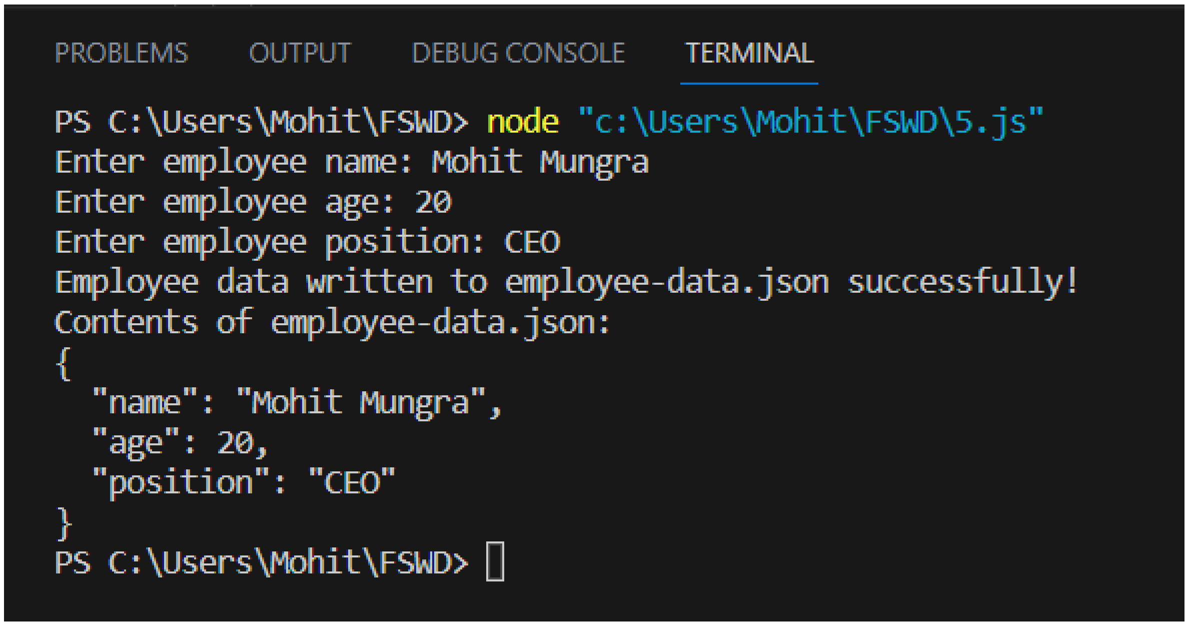
Task-5

Aim :
Read Employee Information from User and Write Data to file called ‘employee-data.json’

Source Code:

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

const rl = readline.createInterface({
  input: process.stdin,
  output: process.stdout
});

// Prompt the user for employee information
rl.question('Enter employee name: ', (name) => {
  rl.question('Enter employee age: ', (age) => {
    rl.question('Enter employee position: ', (position) => {
      // Create an employee object
      const employee = {
        name: name,
        age: parseInt(age),
        position: position
      };

      // Convert the employee object to JSON format
      const employeeJSON = JSON.stringify(employee, null, 2);

      // Write employee data to file
      fs.writeFile('employee-data.json', employeeJSON, (err) => {
        if (err) {
          console.error('Error writing file:', err);
        } else {
          console.log('Employee data written to employee-data.json successfully!');
          // Read and print the contents of the JSON file
          fs.readFile('employee-data.json', 'utf8', (err, data) => {
            if (err) {
              console.error('Error reading file:', err);
            } else {
              console.log('Contents of employee-data.json:');
              console.log(data);
            }
            rl.close();
          });
        }
      });
    });
  });
});
```

Output:



```
PS C:\Users\Mohit\FSWD> node "c:\Users\Mohit\FSWD\5.js"
Enter employee name: Mohit Mungra
Enter employee age: 20
Enter employee position: CEO
Employee data written to employee-data.json successfully!
Contents of employee-data.json:
{
  "name": "Mohit Mungra",
  "age": 20,
  "position": "CEO"
}
PS C:\Users\Mohit\FSWD> 
```

Theoretical Background:

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

These lines import the required modules: readline for reading user input and fs for file system operations in Node.js.

```
2) const rl = readline.createInterface({
 input: process.stdin,
 output: process.stdout
});
```

This code creates an instance of readline.Interface using readline.createInterface(). It sets process.stdin as the input stream and process.stdout as the output stream. This allows reading input from the user and displaying prompts on the command line.

```
3)rl.question('Enter employee name: ', (name) => {
 rl.question('Enter employee age: ', (age) => {
  rl.question('Enter employee position: ', (position) => {
   // Create an employee object
   const employee = {
    name: name,
    age: parseInt(age),
    position: position
   };
```

These lines use the rl.question() method to prompt the user for employee information. The user's inputs for name, age, and position are captured in the respective callback functions (name) => { ... }, (age) => { ... }, and (position) => { ... }.

The code then creates an employee object using the captured information, assigning the user's input for name to the name property, parsing the user's input for age as an integer and assigning it to the age property, and assigning the user's input for position to the position property.

```
4) const employeeJSON = JSON.stringify(employee, null, 2);
```

This line converts the employee object to a JSON string using JSON.stringify(). The second argument (null) specifies no custom formatting options, and the third argument (2) indicates the number of spaces for indentation to make the JSON output more readable.

5)
```
fs.writeFile('employee-data.json', employeeJSON, (err) => {
    if (err) {
      console.error('Error writing file:', err);
    } else {
      console.log('Employee data written to employee-data.json successfully!');
      fs.readFile('employee-data.json', 'utf8', (err, data) => {
        if (err) {
          console.error('Error reading file:', err);
        } else {
          console.log('Contents of employee-data.json:');
          console.log(data);
        }
        rl.close();
      });
    }
  });
```

These lines write the employeeJSON data to the 'employee-data.json' file using fs.writeFile(). If an error occurs, it is logged to the console. If the write operation is successful, it logs a success message.

Then, it reads the contents of the 'employee-data.json' file using fs.readFile(). If an error occurs, it is logged to the console. If the read operation is successful, it logs the contents of the file to the console.

Finally, the rl.close() method is called to close the readline interface.

This code allows the user to enter employee information, writes the data to a JSON file, reads the file contents, and prints them to the console.

Let me know if you need further clarification on any specific part!

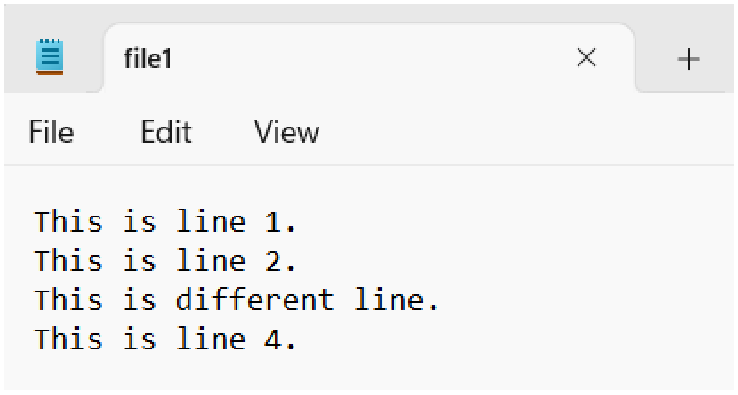
Task-6

Aim :
Compare Two file and show which file is larger and which lines are different

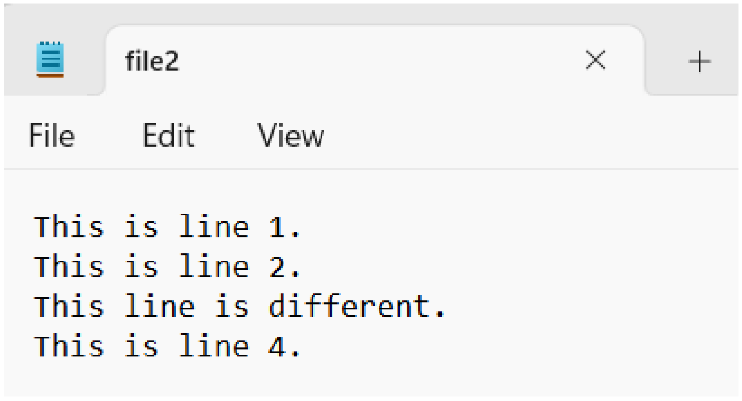
Source Code:

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

try {
  // Read the contents of the first file
  const file1 = fs.readFileSync('file1.txt', 'utf8');

  // Read the contents of the second file
  const file2 = fs.readFileSync('file2.txt', 'utf8');

  // Compare the sizes of the files
  const file1Size = Buffer.byteLength(file1, 'utf8');
  const file2Size = Buffer.byteLength(file2, 'utf8');

  if (file1Size > file2Size) {
    console.log('File 1 is larger than File 2');
  } else if (file1Size < file2Size) {
    console.log('File 2 is larger than File 1');
  } else {
    console.log('File 1 and File 2 have the same size');
  }

  // Split the contents of the files into lines
  const linesFile1 = file1.split('\n');
  const linesFile2 = file2.split('\n');

  // Compare the lines of the files
  for (let i = 0; i < linesFile1.length; i++) {
    if (linesFile1[i] !== linesFile2[i]) {
      console.log(`Line ${i + 1}:`);
      console.log(`File 1: ${linesFile1[i]}`);
      console.log(`File 2: ${linesFile2[i]}`);
      console.log('------------------');
    }
  }
} catch (err) {
  console.error('Error reading the files:', err);
}
```

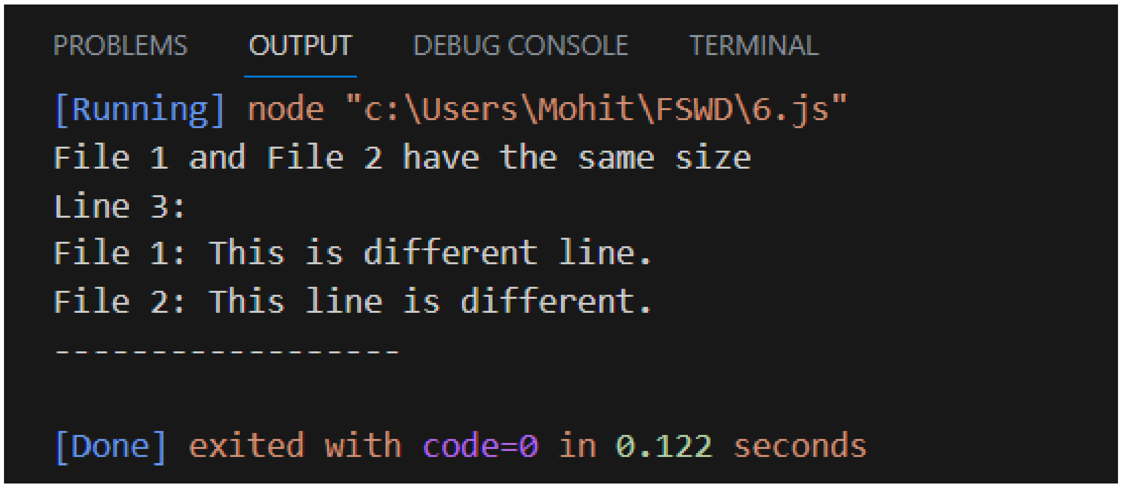
Output:



```
This is line 1.
This is line 2.
This is different line.
This is line 4.
```



```
This is line 1.
This is line 2.
This line is different.
This is line 4.
```



```
[Running] node "c:\Users\Mohit\FSWD\6.js"
File 1 and File 2 have the same size
Line 3:
File 1: This is different line.
File 2: This line is different.
------------------

[Done] exited with code=0 in 0.122 seconds
```

Theoretical Background:

```
1)// Compare the sizes of the files
  const file1Size = Buffer.byteLength(file1, 'utf8');
  const file2Size = Buffer.byteLength(file2, 'utf8');

  if (file1Size > file2Size) {
    console.log('File 1 is larger than File 2');
  } else if (file1Size < file2Size) {
    console.log('File 2 is larger than File 1');
  } else {
    console.log('File 1 and File 2 have the same size');
  }
```

These lines compare the sizes of the files using Buffer.byteLength(). It calculates the byte length of the file contents and compares the sizes. It then prints a message indicating which file is larger or if they have the same size.

```
2)  // Split the contents of the files into lines
  const linesFile1 = file1.split('\n');
  const linesFile2 = file2.split('\n');
```

These lines split the contents of the files into lines by using the split() method and the newline character '\n'. It creates arrays linesFile1 and linesFile2, where each element represents a line of text from the respective files.

```
3)  // Compare the lines of the files
  for (let i = 0; i < linesFile1.length; i++) {
    if (linesFile1[i] !== linesFile2[i]) {
      console.log(`Line ${i + 1}:`);
      console.log(`File 1: ${linesFile1[i]}`);
      console.log(`File 2: ${linesFile2[i]}`);
      console.log('------------------');
    }
  }
} catch (err) {
  console.error('Error reading the files:', err);
}
```

These lines compare the lines of the files using a for loop. It iterates over each line index and checks if the lines at the corresponding indices in linesFile1 and linesFile2 are different. If they are different, it prints the line number, along with the lines from each file. It also adds a separation line for clarity.

The code is wrapped in a try-catch block to catch any errors that may occur during file reading. If an error occurs, it is caught in the catch block, and an error message is printed to the console.

Please make sure to have the files 'file1.txt' and 'file2.txt' available in the same directory as the script or provide the correct file paths to read the desired files.

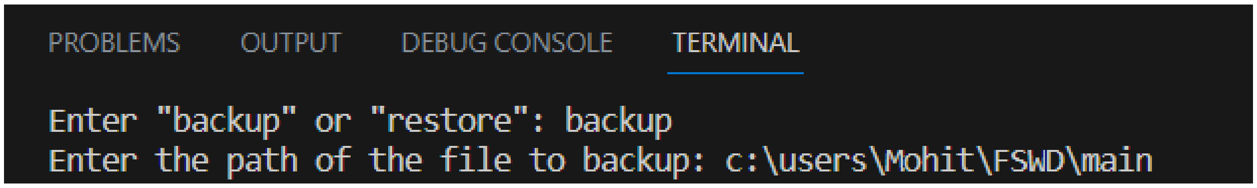
Task-7

Aim :
Create File Backup and Restore Utility

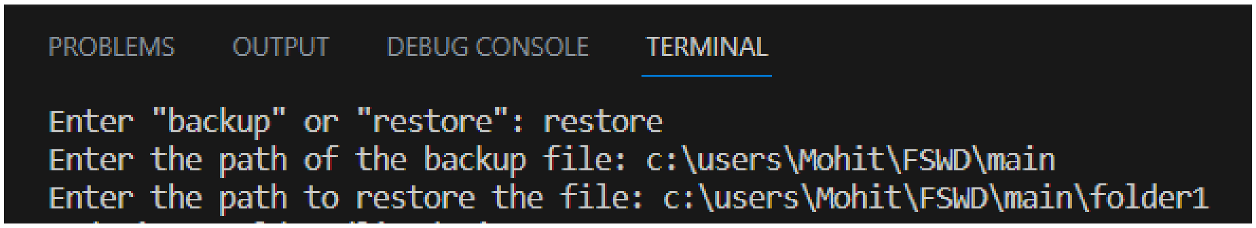
Source Code:

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

const rl = readline.createInterface({
  input: process.stdin,
  output: process.stdout
});

// Function to create a backup of a file
function backupFile(filePath) {
  const fileContent = fs.readFileSync(filePath, 'utf8');
  const backupFileName = path.basename(filePath) + '.bak';
  const backupFilePath = path.join(path.dirname(filePath), backupFileName);
  fs.writeFileSync(backupFilePath, fileContent);
  console.log(`Backup created: ${backupFilePath}`);
}

// Function to restore a file from backup
function restoreFile(backupFilePath, originalFilePath) {
  const fileContent = fs.readFileSync(backupFilePath, 'utf8');
  fs.writeFileSync(originalFilePath, fileContent);
  console.log(`File restored: ${originalFilePath}`);
}

// Prompt the user for backup or restore
rl.question('Enter "backup" or "restore": ', (choice) => {
  if (choice.toLowerCase() === 'backup') {
    rl.question('Enter the path of the file to backup: ', (filePath) => {
      backupFile(filePath);
      rl.close();
    });
  } else if (choice.toLowerCase() === 'restore') {
    rl.question('Enter the path of the backup file: ', (backupFilePath) => {
      rl.question('Enter the path to restore the file: ', (originalFilePath) => {
        restoreFile(backupFilePath, originalFilePath);
        rl.close();
      });
    });
  } else {
    console.log('Invalid choice. Please enter either "backup" or "restore".');
    rl.close();
  }
});
```

Output:



```
Enter "backup" or "restore": backup
Enter the path of the file to backup: c:\users\Mohit\FSWD\main
```



```
Enter "backup" or "restore": restore
Enter the path of the backup file: c:\users\Mohit\FSWD\main
Enter the path to restore the file: c:\users\Mohit\FSWD\main\folder1
```

Theoretical Background:

1)
```
const rl = readline.createInterface({
  input: process.stdin,
  output: process.stdout
});
```

This creates an instance of readline.Interface for reading input from the user via the standard input (process.stdin) and displaying output via the standard output (process.stdout).

2)
```
function backupFile(filePath) {
  const fileContent = fs.readFileSync(filePath, 'utf8');
  const backupFileName = path.basename(filePath) + '.bak';
  const backupFilePath = path.join(path.dirname(filePath), backupFileName);
  fs.writeFileSync(backupFilePath, fileContent);
  console.log(`Backup created: ${backupFilePath}`);
}
```

This defines a function backupFile that takes a filePath as an argument. It reads the content of the file using fs.readFileSync, creates a backup file name by appending .bak to the original file name using path.basename and path.join, and writes the file content to the backup file using fs.writeFileSync.

3)
```
function restoreFile(backupFilePath, originalFilePath) {
  const fileContent = fs.readFileSync(backupFilePath, 'utf8');
  fs.writeFileSync(originalFilePath, fileContent);
  console.log(`File restored: ${originalFilePath}`);
}
```

This defines a function restoreFile that takes backupFilePath and originalFilePath as arguments. It reads the content of the backup file using fs.readFileSync and writes the content to the original file using fs.writeFileSync, effectively restoring the file from the backup.

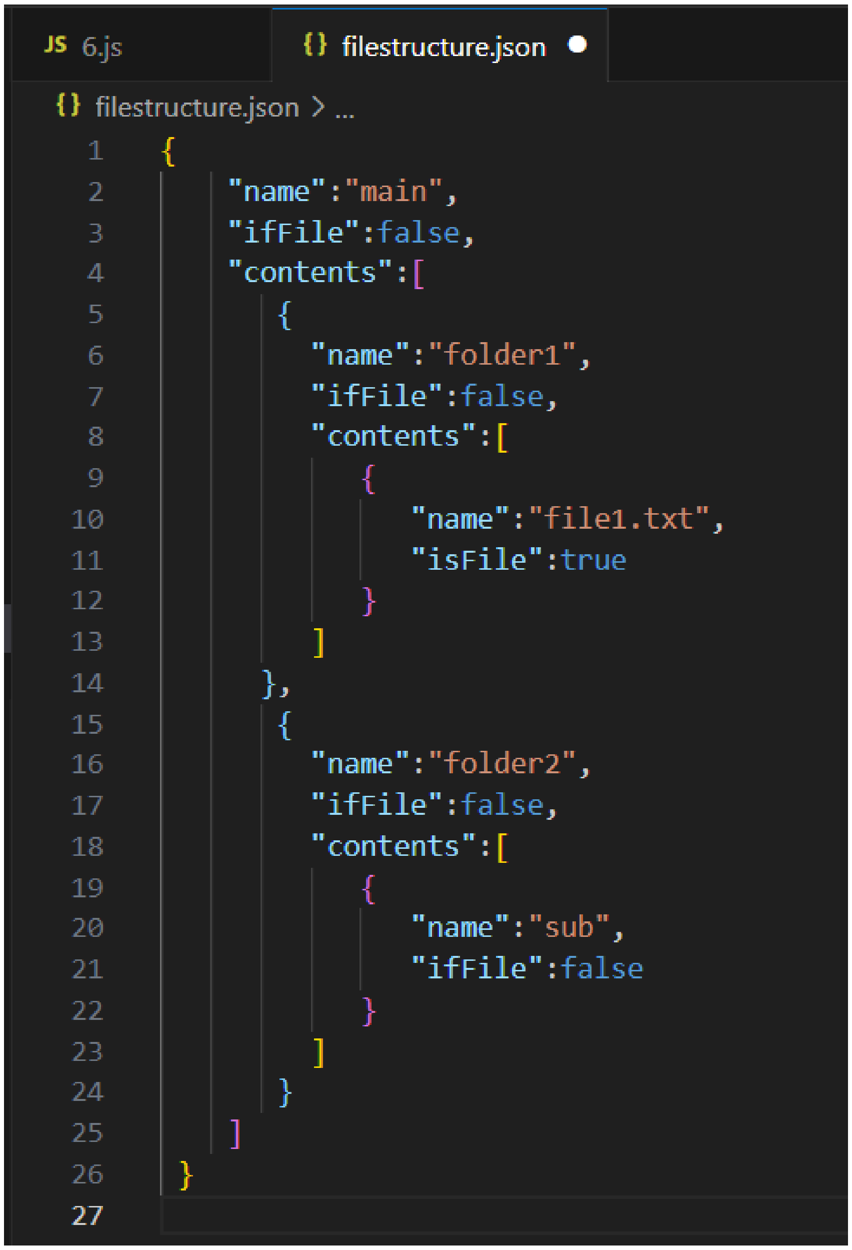
Task-8

Aim :
Create File/Folder Structure given in json file.

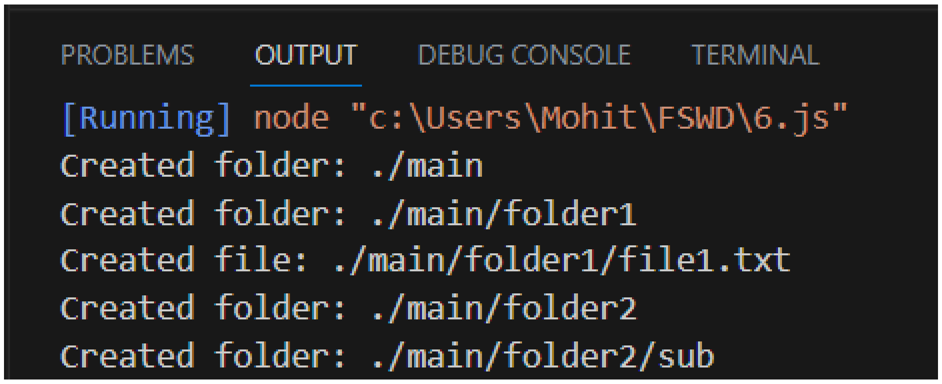
Source Code:

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

function createFileStructure(basePath, structure) {
  if (structure.isFile) {
    fs.writeFileSync(`${basePath}/${structure.name}`, '');
    console.log(`Created file: ${basePath}/${structure.name}`);
  } else {
    fs.mkdirSync(`${basePath}/${structure.name}`);
    console.log(`Created folder: ${basePath}/${structure.name}`);
    for (const item of structure.contents) {
      createFileStructure(`${basePath}/${structure.name}`, item);
    }
  }
}

const jsonContent = fs.readFileSync('fileStructure.json', 'utf8');
const fileStructure = JSON.parse(jsonContent);

createFileStructure('.', fileStructure);
```

Output:



```
{
    "name":"main",
    "ifFile":false,
    "contents":[
      {
        "name":"folder1",
        "ifFile":false,
        "contents":[
          {
            "name":"file1.txt",
            "isFile":true
          }
        ]
     },
      {
        "name":"folder2",
        "ifFile":false,
        "contents":[
          {
            "name":"sub",
            "ifFile":false
          }
        ]
      }
    ]
 }
```



```
[Running] node "c:\Users\Mohit\FSWD\6.js"
Created folder: ./main
Created folder: ./main/folder1
Created file: ./main/folder1/file1.txt
Created folder: ./main/folder2
Created folder: ./main/folder2/sub
```

Theoretical Background:

1)
```
  if (structure.isFile) {
    fs.writeFileSync(`${basePath}/${structure.name}`, '');
    console.log(`Created file: ${basePath}/${structure.name}`);
  } else {
    fs.mkdirSync(`${basePath}/${structure.name}`);
    console.log(`Created folder: ${basePath}/${structure.name}`);
    for (const item of structure.contents) {
      createFileStructure(`${basePath}/${structure.name}`, item);
    }
  }
}
```

This block of code checks if the current item in the structure is a file or a folder using the isFile property. If it's a file, it creates an empty file using fs.writeFileSync, specifying the path based on the basePath and structure.name. It then logs a message indicating the creation of the file.

If the current item is a folder, it creates a directory using fs.mkdirSync, specifying the path based on the basePath and structure.name. It then logs a message indicating the creation of the folder.

It then iterates over the structure.contents array and recursively calls the createFileStructure function for each nested item, passing the updated path (${basePath}/${structure.name}) and the current nested item.

2)
```
const jsonContent = fs.readFileSync('fileStructure.json', 'utf8');
  const fileStructure = JSON.parse(jsonContent);
```
These lines read the content of the JSON file fileStructure.json using fs.readFileSync and store it in the jsonContent variable. Then, it parses the JSON content into a JavaScript object using JSON.parse and assigns it to the fileStructure variable.

3)
```
createFileStructure('.', fileStructure);
```
This line calls the createFileStructure function with the base path . (representing the current directory) and the fileStructure object to create the file/folder structure.
Make sure to have the JSON file (fileStructure.json) available in the same directory as the script, and adjust the file path and structure according to your requirements.

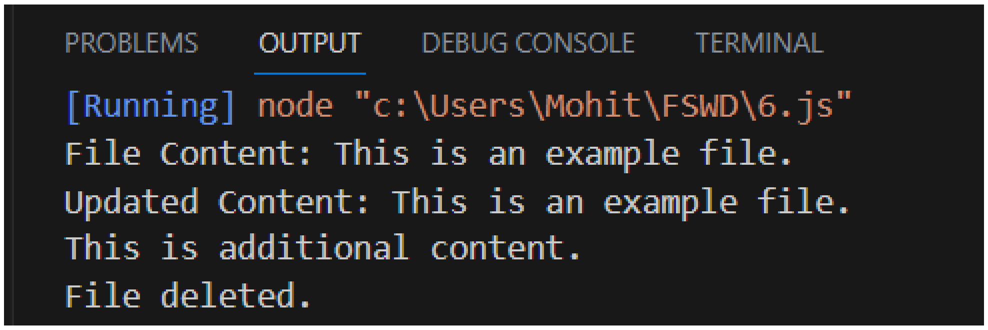
Task-9

Aim :
Experiment with : Create File,Read File,Append File,Delete File,Rename File,List Files/Dirs

Source Code:

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

// Create a file
fs.writeFileSync('example.txt', 'This is an example file.');

// Read a file
const fileContent = fs.readFileSync('example.txt', 'utf8');
console.log('File Content:', fileContent);

// Append to a file
fs.appendFileSync('example.txt', '\nThis is additional content.');

// Read the updated file
const updatedContent = fs.readFileSync('example.txt', 'utf8');
console.log('Updated Content:', updatedContent);

// Delete a file
fs.unlinkSync('example.txt');
console.log('File deleted.');

// Rename a file
fs.renameSync('example.txt', 'renamed.txt');
console.log('File renamed.');

// List files and directories
const dirPath = '.';
const filesAndDirs = fs.readdirSync(dirPath);
console.log('Files and Directories:');
filesAndDirs.forEach((item) => {
  const fullPath = path.join(dirPath, item);
  const stats = fs.statSync(fullPath);
  if (stats.isDirectory()) {
    console.log('Directory:', item);
  } else {
    console.log('File:', item);
  }
});
```

Output:



```
[Running] node "c:\Users\Mohit\FSWD\6.js"
File Content: This is an example file.
Updated Content: This is an example file.
This is additional content.
File deleted.
```

Theoretical Background:
1) fs.writeFileSync('example.txt', 'This is an example file.');
This line creates a new file named 'example.txt' and writes the content 'This is an example file.' into it using fs.writeFileSync

2)const fileContent = fs.readFileSync('example.txt', 'utf8');
console.log('File Content:', fileContent);
This code reads the content of the file 'example.txt' synchronously using fs.readFileSync and stores it in the fileContent variable. It then logs the content to the console.

3)fs.appendFileSync('example.txt', '\nThis is additional content.');
This line appends the string '\nThis is additional content.' to the existing content of the 'example.txt' file using fs.appendFileSync.

4)const updatedContent = fs.readFileSync('example.txt', 'utf8');
console.log('Updated Content:', updatedContent);
This code reads the updated content of the 'example.txt' file using fs.readFileSync and stores it in the updatedContent variable. It then logs the updated content to the console.

5)fs.unlinkSync('example.txt');
console.log('File deleted.');

This line deletes the file 'example.txt' using fs.unlinkSync and logs a message indicating that the file has been deleted.

6)fs.renameSync('example.txt', 'renamed.txt');
console.log('File renamed.');

This line renames the file 'example.txt' to 'renamed.txt' using fs.renameSync and logs a message indicating that the file has been renamed.

Learning Outcome :

CO1 : Understand various technologies and trends impacting single page web applications.
CO4 : Demonstrate the use of JavaScript to fulfill the essentials of front-end development To back-end development.