

Week-2(Node.js)

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('Post Method')
  }
});

httpserver.listen(4000, ()=>{
  console.log('Listening on on port 4000');
})
```

Output:

```
PS C:\STUDY\Sem 5\FSWD\week-2 Node.js> node task1.js
Listening on on port 4000
█
```

Theoretical Background:

1 - http module import:

- The code starts by importing the http module, which is a built-in module in Node.js that provides functionality to create HTTP servers and make HTTP requests.

2 - Create an HTTP server:

- The code creates an HTTP server using the `http.createServer()` method. The server is created with a callback function that will be invoked whenever a request is received by the server.

3 - Request handling:

- Inside the callback function, the code checks if the incoming request method is POST using the condition `if (req.method = 'POST')`. It is important to note that there's a typo in the code, and the single equals sign (`=`) is used instead of the comparison operator (`===`). This typo will affect the expected behavior of the server.

4 - Sending response:

- If the request method is indeed POST, the server sends the response "Post Method" using `res.end('Post Method')`. The `res.end()` method is used to end the response and send the specified data to the client.

5 - Server listening:

- After defining the server behavior, the code uses the `httpserver.listen()` method to start listening on port 4000. Once the server is up and running, it logs a message "Listening on port 4000" to the console.

Task-2

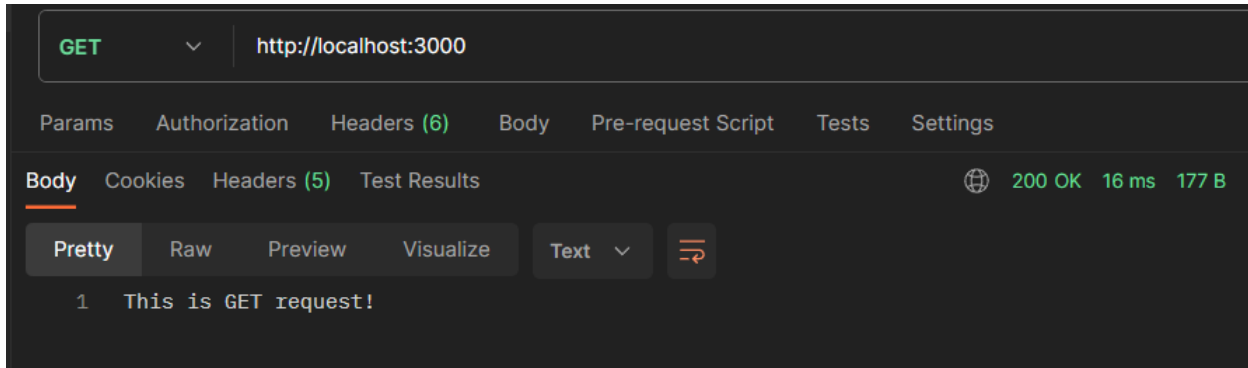
Aim :

Experiment with Various HTTP Methods, Content Types and Status Code

Source Code:**HTTP Methods:**

```
const http = require('http');
const server = http.createServer((req, res) => {
  if (req.method === 'GET') {
    res.writeHead(200, { 'Content-Type': 'text/plain' });
    res.end('This is GET request!');
  } else if (req.method === 'POST') {
    res.writeHead(200, { 'Content-Type': 'text/plain' });
    res.end('This is POST request!');
  } else if (req.method === 'PUT') {
    res.writeHead(200, { 'Content-Type': 'text/plain' });
    res.end('This is PUT request!');
  } else if (req.method === 'DELETE') {
    res.writeHead(200, { 'Content-Type': 'text/plain' });
    res.end('This is DELETE request!');
  } else if (req.method === 'PATCH') {
    res.writeHead(200, { 'Content-Type': 'text/plain' });
    res.end('This is Patch request!');
  } else {
    res.writeHead(405, { 'Content-Type': 'text/plain' });
    res.end('Method Not Allowed');
  }
});

const port = 3000;
server.listen(port, () => {
  console.log(`Server is listening on http://localhost:${port}`);
});
```

Output:**HTTP Methods:****GET:**

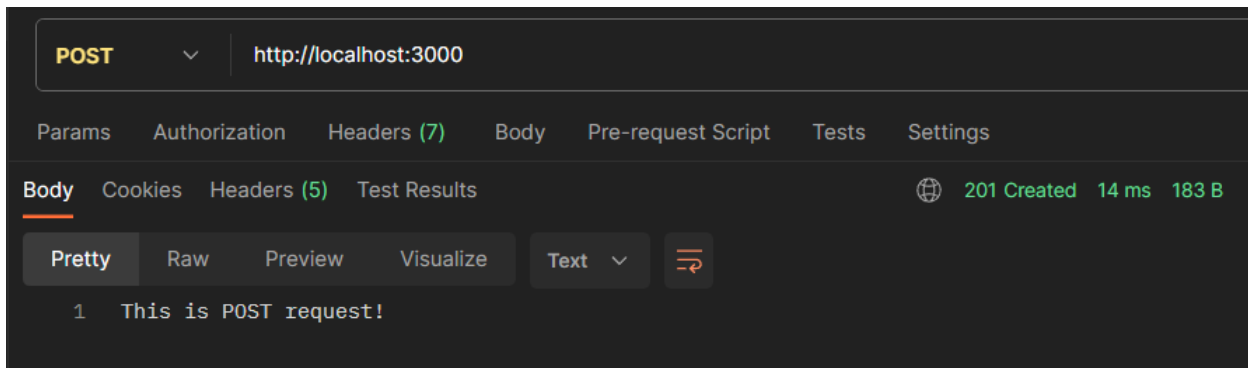
GET http://localhost:3000

Params Authorization Headers (6) Body Pre-request Script Tests Settings

Body Cookies Headers (5) Test Results 200 OK 16 ms 177 B

Pretty Raw Preview Visualize Text

1 This is GET request!

POST:

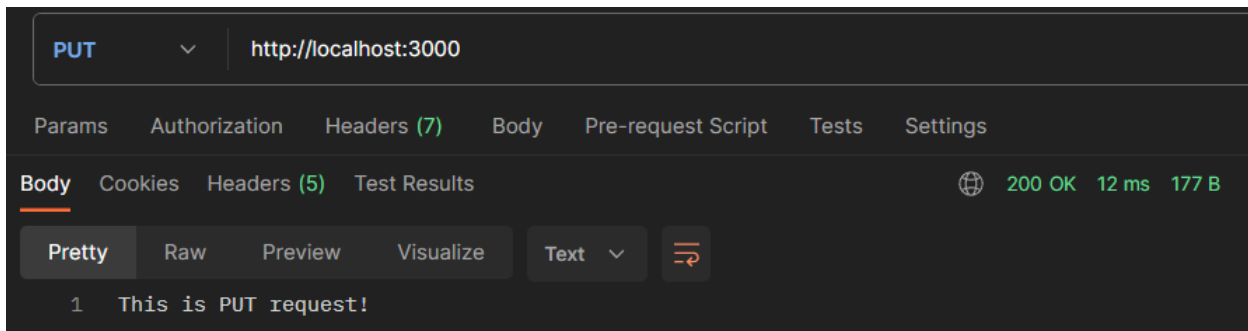
POST http://localhost:3000

Params Authorization Headers (7) Body Pre-request Script Tests Settings

Body Cookies Headers (5) Test Results 201 Created 14 ms 183 B

Pretty Raw Preview Visualize Text

1 This is POST request!

PUT:

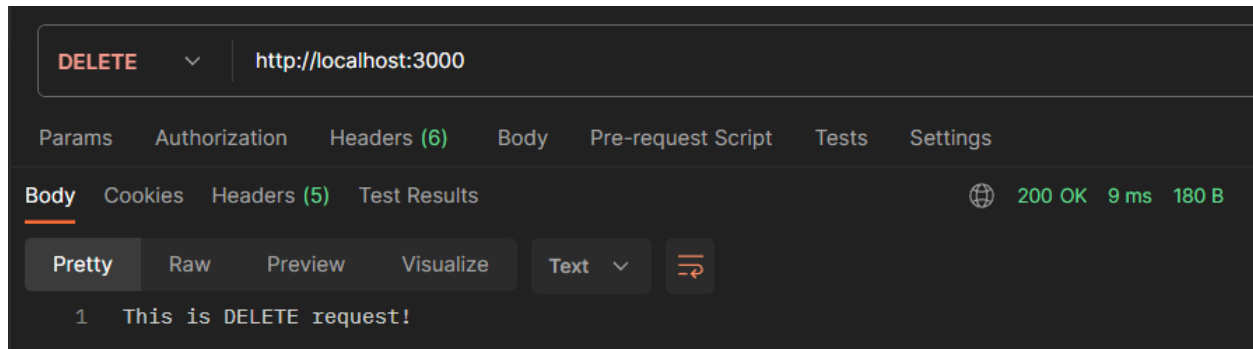
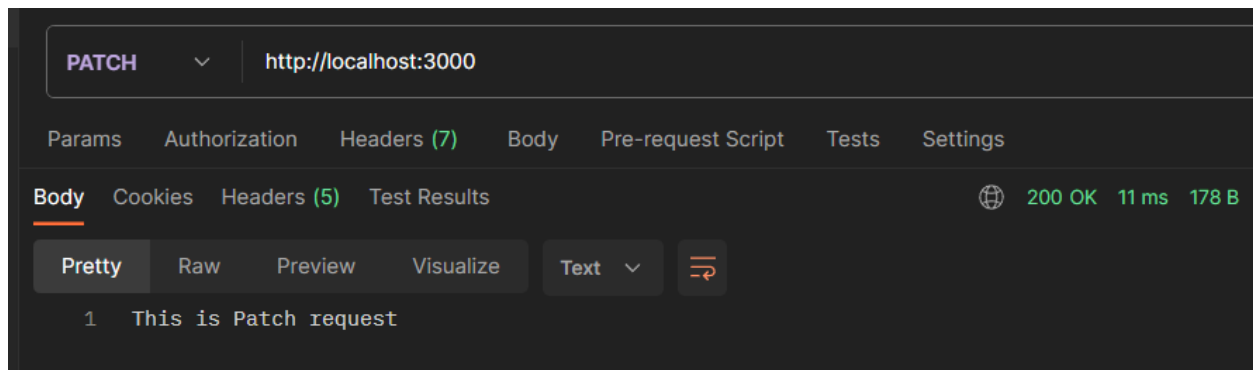
PUT http://localhost:3000

Params Authorization Headers (7) Body Pre-request Script Tests Settings

Body Cookies Headers (5) Test Results 200 OK 12 ms 177 B

Pretty Raw Preview Visualize Text

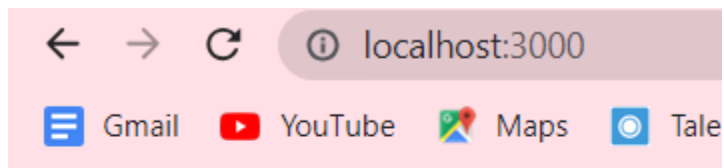
1 This is PUT request!

DELETE:**PATCH:**

Source Code:**Content Type:**

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

http.createServer((req, res) => {
  const readStream = fs.createReadStream('test.html')
  res.writeHead(200, { 'content-type': 'text/html' })
  readStream.pipe(res)
}).listen(3000);
```

Output:

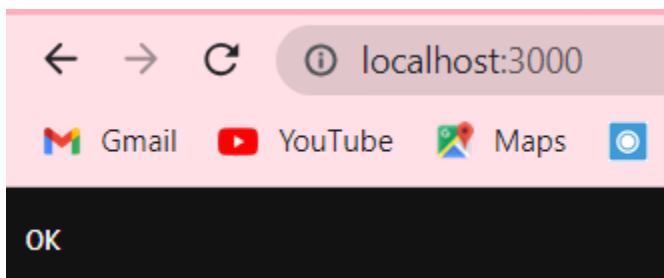
HELLO!!

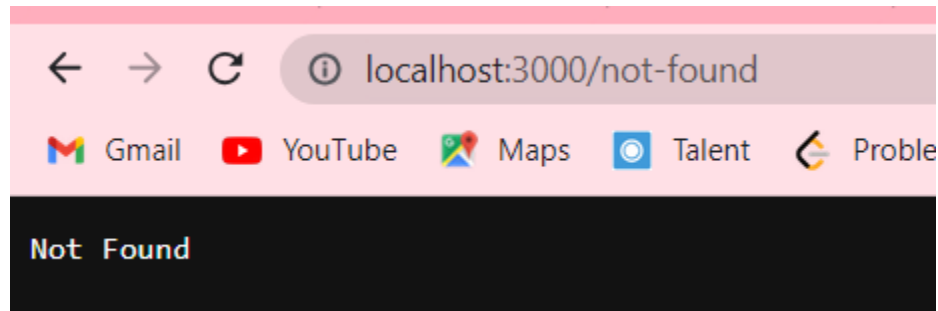
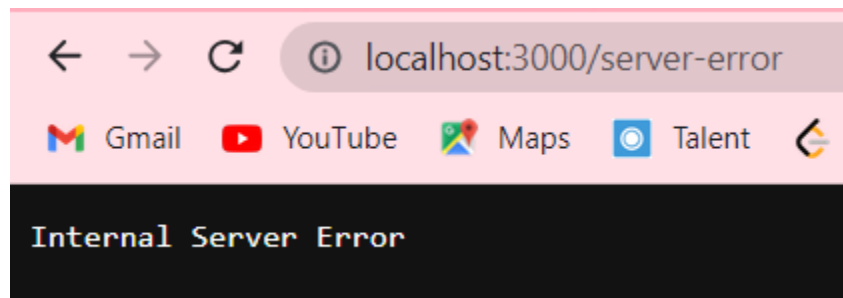
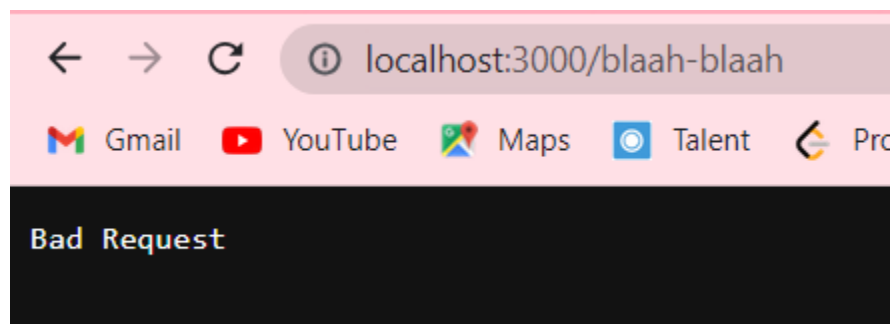
Source Code:**Status Code**

```
const http = require('http');
const server = http.createServer((req, res) => {

  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**

404**500****400**

Theoretical Background:

1 - HTTP Methods :

- Common HTTP methods include GET, POST, PUT, DELETE, and more. Each technique has a distinct use and goal:
- Information can be retrieved from a server using **GET**.
- Data is sent via **POST** to the server in order to build new resources.
- **PUT** is used to update or replace existing resources by sending data to the server.
- **DELETE** is the command used to ask the server to delete a resource.

2 - Content Types:

- To identify the type of material being delivered or received, HTTP requests and replies frequently include a Content-Type header. Application/json, application/xml, text/html, multipart, and form-data are examples of common content types.
- By adjusting the Content-Type header in your requests, you can experiment with various content kinds.

3 - Status Codes:

Information about the outcome of an HTTP request is provided via HTTP status codes. Each status code denotes a certain circumstance or state. For example:

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

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('student-data.txt', 'utf8', (err, data) => {
  if (err) {
    console.error('Error reading the file:', err);
    return;
  }

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

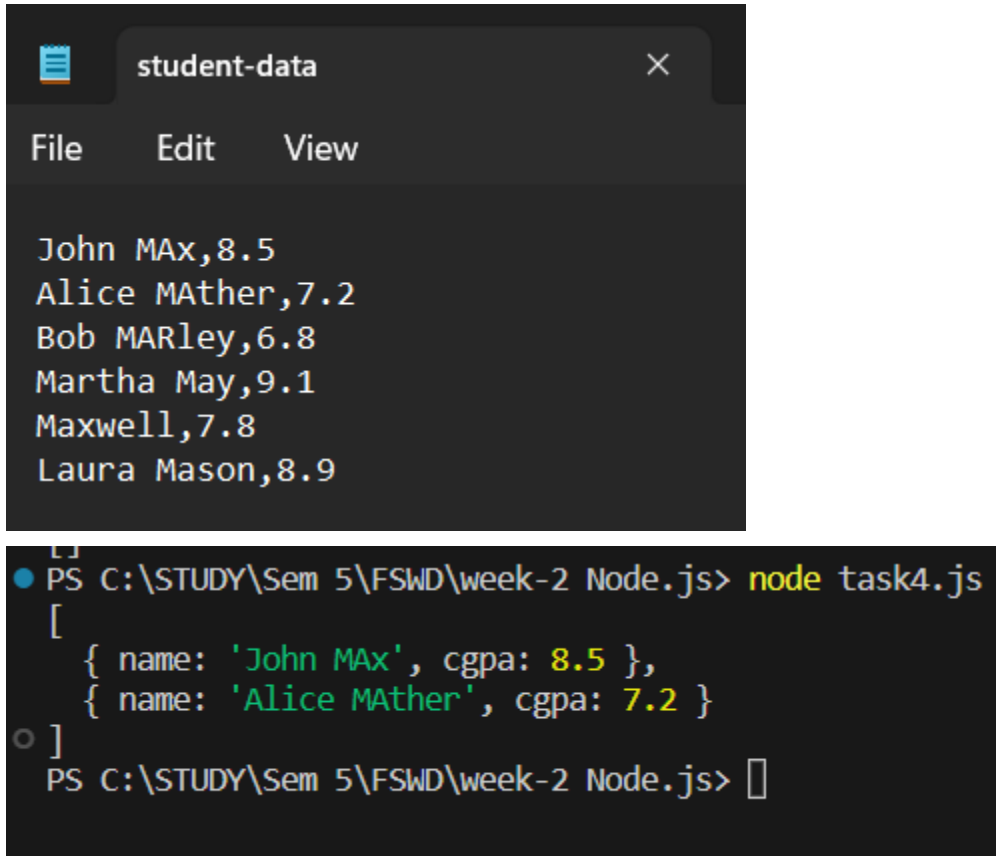
  const filteredStudents = [];

  lines.forEach((line) => {

    const [name, cgpa] = line.split(',');
    if (name.includes('MA') && parseFloat(cgpa) > 7) {

      filteredStudents.push({ name, cgpa: parseFloat(cgpa) });
    }
  });

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

Output:

The image shows two screenshots. The top screenshot is a code editor window titled 'student-data' with a menu bar (File, Edit, View) and a list of student names and CGPAs: John MAX,8.5; Alice Mather,7.2; Bob MARley,6.8; Martha May,9.1; Maxwell,7.8; and Laura Mason,8.9. The bottom screenshot is a terminal window showing the command 'node task4.js' being executed, which outputs a JSON array of two student objects: [{ name: 'John MAX', cgpa: 8.5 }, { name: 'Alice Mather', cgpa: 7.2 }].

```
student-data
File Edit View

John MAX,8.5
Alice Mather,7.2
Bob MARley,6.8
Martha May,9.1
Maxwell,7.8
Laura Mason,8.9

PS C:\STUDY\Sem 5\FSWD\week-2 Node.js> node task4.js
[
  { name: 'John MAX', cgpa: 8.5 },
  { name: 'Alice Mather', cgpa: 7.2 }
]
PS C:\STUDY\Sem 5\FSWD\week-2 Node.js> 
```

Theoretical Background:

- The code uses the fs module to read the contents of the file student-data.txt using the fs.readFile method. It reads the data in UTF-8 encoding.
- After reading the file, the callback function is executed. If there is an error during file reading, the error is logged to the console.
- Assuming the file is read successfully, the data is split into an array of lines using data.split('\n'). Each line in the file represents information about a student, with each line being a comma-separated value (CSV) containing the student's name and CGPA (Cumulative Grade Point Average).
- The code initializes an empty array named filteredStudents, which will hold the student records that meet specific criteria.
- The code then uses a forEach loop to iterate over each line in the lines array.
- For each line, the line's content is split into name and cgpa using destructuring assignment with line.split(',').
- The code checks whether the name includes the string 'MA' (assuming 'MA' is part of the student's name, e.g., a course or major designation) and whether the cgpa (parsed as a float) is greater than 7. If both conditions are true, the student's record is considered to meet the filter criteria.

- If a student's record meets the filter criteria, an object with the name and cgpa properties is created and pushed into the filteredStudents array.
- After processing all the lines, the script outputs the filteredStudents array to the console, displaying the student records that meet the filter criteria.

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
});

rl.question('Enter employee name: ', (name) => {
  rl.question('Enter employee age: ', (age) => {
    rl.question('Enter employee position: ', (position) => {

      const employee = {
        name: name,
        age: parseInt(age),
        position: position
      };

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

      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();  
  });  
}  
});  
});  
});  
});
```

Output:

```
PS C:\STUDY\Sem 5\FSWD\week-2 Node.js> node task5.js  
Enter employee name: Akshar  
Enter employee age: 19  
Enter employee position: Owner  
Employee data written to employee-data.json successfully!  
Contents of employee-data.json:  
{  
  "name": "Akshar",  
  "age": 19,  
  "position": "Owner"  
}  
PS C:\STUDY\Sem 5\FSWD\week-2 Node.js> █
```

Theoretical Background:

- The script requires two built-in Node.js modules, readline and fs. The readline module provides an interface for reading input from the command line, while the fs module allows reading and writing to the file system.
- The script creates an instance of the readline interface with readline.createInterface. It sets the input to process.stdin (command-line input) and the output to process.stdout (command-line output).
- The script uses the rl.question method to prompt the user to enter the employee's name, age, and position, one by one. It uses nested callbacks to collect this information in sequence.
- After collecting the employee information, it creates an employee object containing the entered details and converts it to a formatted JSON string using JSON.stringify.

- The `fs.writeFile` method is then used to write the employee JSON data to the `employee-data.json` file. If there is an error during the writing process, it is logged to the console.
- If the file write operation is successful, it displays a success message confirming that the data has been written to the file.
- Next, the script uses `fs.readFile` to read the content of the `employee-data.json` file in UTF-8 encoding. It also uses a callback to handle any errors that may occur during the reading process.
- If the file reading is successful, it displays the contents of the `employee-data.json` file, showing the JSON data representing the employee information.
- Finally, the script closes the readline interface using `rl.close()`.

Task-6

Aim :

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

Source Code:

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

function compareFiles(file1, file2) {
  const data1 = fs.readFileSync(file1, 'utf8').split('\n');
  const data2 = fs.readFileSync(file2, 'utf8').split('\n');

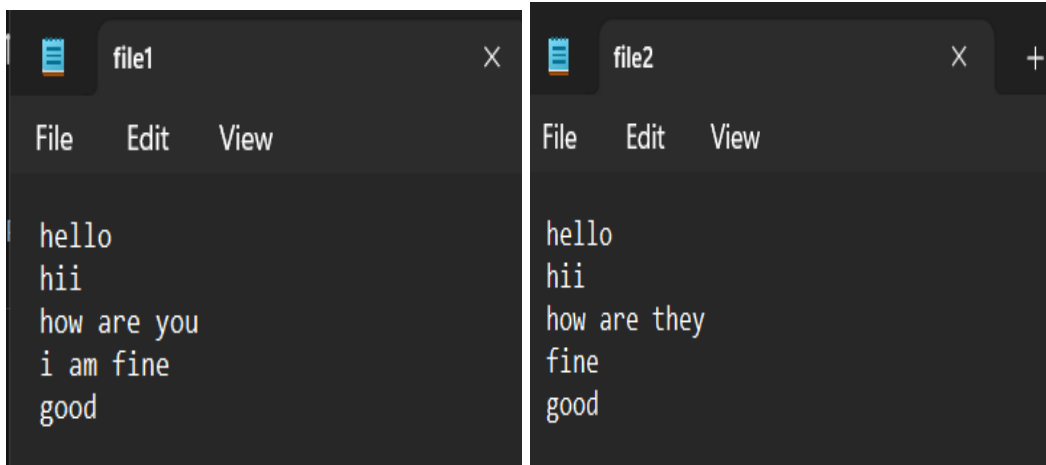
  // Compare file sizes
  const fileSize1 = fs.statSync(file1).size;
  const fileSize2 = fs.statSync(file2).size;

  console.log(`File '${file1}' size: ${fileSize1} bytes`);
  console.log(`File '${file2}' size: ${fileSize2} bytes`);

  // Compare and print different lines
  console.log('\nLines with differences:');
  for (let i = 0; i < Math.max(data1.length, data2.length); i++) {
    if (data1[i] !== data2[i]) {
      const lineNum = i + 1;
      const line1 = data1[i] || '';
      const line2 = data2[i] || '';
    }
  }
}
```

```
        console.log(`Line ${lineNum}:`);
        console.log(`File '${file1}': ${line1}`);
        console.log(`File '${file2}': ${line2}`);
        console.log('---');
    }
}

// Usage
const file1 = 'file1.txt';
const file2 = 'file2.txt';
compareFiles(file1, file2);
```

Output:


```
PS C:\STUDY\Sem 5\FSWD\week-2 Node.js> node task6.js
File 'file1.txt' size: 40 bytes
File 'file2.txt' size: 37 bytes

Lines with differences:
Line 3:
File 'file1.txt': how are you
File 'file2.txt': how are they
---
Line 4:
File 'file1.txt': i am fine
File 'file2.txt': fine
---
PS C:\STUDY\Sem 5\FSWD\week-2 Node.js> █
```

Theoretical Background:

- The code uses the fs module to read the content of file1 and file2 using the fs.readFileSync method. The data is read in UTF-8 encoding and split into arrays of lines using split('\n').
- The sizes of both files (file1 and file2) are obtained using fs.statSync(file1).size and fs.statSync(file2).size, respectively.
- The script outputs the sizes of both files to the console.
- The code then compares the contents of both files line by line. It iterates over the lines of both files using a for loop. The loop continues until it reaches the end of the longest file, determined by Math.max(data1.length, data2.length).
- For each line index i, the script checks if the lines at index i in both data1 and data2 are different. If they are different, it indicates a difference between the files.
- If a line is different, the script prints the line number (lineNum) and the content of the line in both files (line1 and line2) to the console, indicating the differences between the two files.

Task-7

Aim :

Create File Backup and Restore Utility

Source Code:

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

function createFileBackup(filePath, backupDir) {
  const originalFileName = path.basename(filePath);
  const backupFileName = `${originalFileName}.bak`;

  // Create the backup directory if it doesn't exist
  if (!fs.existsSync(backupDir)) {
    fs.mkdirSync(backupDir);
  }

  // Backup the file
  fs.copyFileSync(filePath, path.join(backupDir, backupFileName));
  console.log(`Backup created: ${backupFileName}`);
}

function restoreFileBackup(filePath, backupDir) {
  const originalFileName = path.basename(filePath);
  const backupFileName = `${originalFileName}.bak`;

  const backupFilePath = path.join(backupDir, backupFileName);

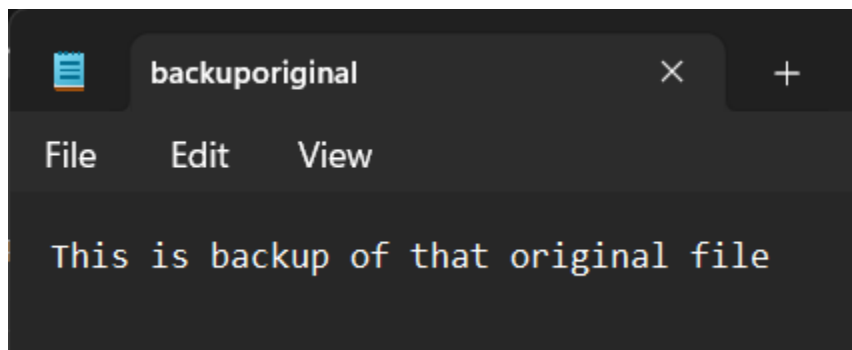
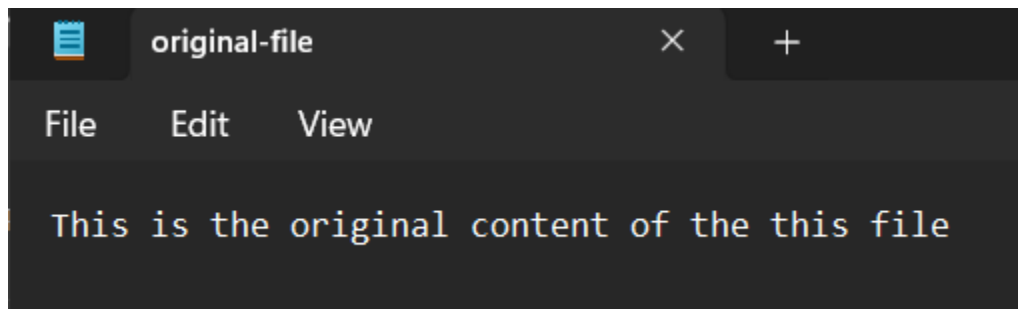
  if (fs.existsSync(backupFilePath)) {
    // Restore the file from backup
    fs.copyFileSync(backupFilePath, filePath);
    console.log(`Backup restored: ${backupFileName}`);
  } else {
    console.log(`Backup file not found: ${backupFileName}`);
  }
}

// Usage
```

```
const originalFilePath = 'original-file.txt';
const backupDirectory = 'backup';

// Create a backup of the original file
createFileBackup(originalFilePath, backupDirectory);

// Restore the original file from the backup
restoreFileBackup(originalFilePath, backupDirectory);
```

Output:

```
PS C:\STUDY\Sem 5\FSWD\week-2 Node.js> node task7.js
Backup created: original-file.txt.bak
Backup restored: original-file.txt.bak
PS C:\STUDY\Sem 5\FSWD\week-2 Node.js> 
```

Theoretical Background:

1 - createFileBackup function:

- The createFileBackup function takes two arguments: filePath, which is the path to the file you want to back up, and backupDir, which is the directory where the backup will be stored.
- It extracts the original file's name from the filePath using path.basename.
- It generates the backup file name by appending the .bak extension to the original file name.
- It checks if the backupDir exists. If not, it creates the backup directory using fs.mkdirSync.
- It then creates the backup by copying the original file to the backup directory using fs.copyFileSync.
- Finally, it outputs a message confirming the successful creation of the backup.

2 - restoreFileBackup function:

- The restoreFileBackup function also takes two arguments: filePath, which is the path to the file you want to restore, and backupDir, which is the directory where the backup is stored.
- It extracts the original file's name from the filePath using path.basename and generates the backup file name by appending the .bak extension to the original file name.
- It constructs the full path to the backup file using path.join.
- It checks if the backup file exists in the specified backup directory using fs.existsSync.
- If the backup file is found, it restores the original file by copying the content of the backup file back to the original file path using fs.copyFileSync.
- If the backup file is not found, it outputs a message indicating that the backup file was not found.

3 - Usage:

- The code demonstrates how to use the two functions by creating a backup of the file specified in originalFilePath and storing it in the backupDirectory.
- After creating the backup, it restores the original file from the backup using the restoreFileBackup function, demonstrating the restoration process.

Task-8

Aim :

Create File/Folder Structure given in json file.

Source Code:

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

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

try {
  const jsonContent = fs.readFileSync('fileStructure.json', 'utf8');
  const fileStructure = JSON.parse(jsonContent);
  createFileStructure('.', fileStructure);
  console.log('File/Folder structure created successfully!');
}
```

```
} catch (error) {  
  console.error('Error reading or parsing JSON file:', error);  
}
```

Output:

```
Created folder: ./root  
Created folder: ./root/folder1  
Created file: ./root/folder1/file1.txt  
Created file: ./root/folder1/file2.txt  
Created folder: ./root/folder2  
Created folder: ./root/folder2/subfolder1  
Created file: ./root/folder2/subfolder1/file3.txt  
Created file: ./root/folder2/file4.txt  
File/Folder structure created successfully!  
PS C:\STUDY\Sem 5\FSWD\week-2 Node.js> █
```

```
{ } fileStructure.json > [ ] contents
1   {
2     "name": "root",
3     "isFile": false,
4     "contents": [
5       {
6         "name": "folder1",
7         "isFile": false,
8         "contents": [
9           {
10            "name": "file1.txt",
11            "isFile": true
12          },
13          {
14            "name": "file2.txt",
15            "isFile": true
16          }
17        ]
18      },
19      {
20        "name": "folder2",
21        "isFile": false,
22        "contents": [
23          {
24            "name": "subfolder1",
25            "isFile": false,
26            "contents": [
27              {
28                "name": "file3.txt",
29                "isFile": true
30              }
31            ]
32          }
33        ]
34      }
35    ]
36  }
```

Theoretical Background:

1 - createFileStructure function:

- The createFileStructure function takes two arguments: basePath, which is the path where the file/folder structure should be created, and structure, which represents the JSON data for the file/folder structure.
- If the structure is marked as a file (property isFile is true), the function attempts to create the file using fs.writeFileSync. If successful, it logs a message indicating that the file has been created. If there is an error during file creation, it logs an error message with the specific file path and the error details.
- If the structure is marked as a folder (property isFile is false), the function attempts to create the folder using fs.mkdirSync. If successful, it logs a message indicating that the folder has been created. If there is an error during folder creation, it logs an error message with the specific folder path and the error details.
- If the folder has a contents property, the function recursively calls itself for each item in the contents array, creating the entire folder structure recursively.

2 - Main Script:

- The main script starts by wrapping the code inside a try-catch block to handle any potential errors during file reading or JSON parsing.
- It reads the content of the fileStructure.json file using fs.readFileSync in UTF-8 encoding and stores the content in the variable jsonContent.
- The jsonContent is then parsed using JSON.parse to obtain the file/folder structure in a JavaScript object format, stored in the variable fileStructure.
- The script then invokes the createFileStructure function with . (current directory) as the basePath and fileStructure as the structure to create the file/folder structure based on the JSON data.
- If the file/folder structure creation is successful, the script logs a success message.

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
const filePath = 'test.txt';
fs.writeFileSync(filePath, 'This is a test file.');
```



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



```
// Append to a file
fs.appendFileSync(filePath, '\n Adding some extra content.');
```



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



```
// Delete a file
fs.unlinkSync(filePath);
console.log('File deleted.');
```



```
// Rename a file
const newFilePath = 'renamed.txt';
fs.renameSync(filePath, newFilePath);
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:

```
PS C:\STUDY\Sem 5\FSWD\week-2 Node.js> node task9.js
File Content: This is a test file.
Updated Content: This is a test file.
Adding some extra content.
File deleted.
```

Theoretical Background:

1 - Create a file:

- The code uses `fs.writeFileSync` to create a new file named `test.txt` with the content "This is a test file."

2 - Read a file:

- The code uses `fs.readFileSync` to read the content of the `test.txt` file in UTF-8 encoding and stores it in the `fileContent` variable. It then logs the content to the console.

3 - Append to a file:

- The code uses `fs.appendFileSync` to add the text "\n Adding some extra content." to the end of the `test.txt` file.

4 - Read the updated file:

- The code uses `fs.readFileSync` again to read the content of the updated `test.txt` file and stores it in the `updatedContent` variable. It then logs the updated content to the console.

5 - Delete a file:

- The code uses `fs.unlinkSync` to delete the `test.txt` file from the file system.

6 - Rename a file:

- The code uses `fs.renameSync` to rename the `test.txt` file to `renamed.txt`.

7 - List files and directories:

- The code uses `fs.readdirSync` to list the contents of the current directory (denoted by `.`). For each item (file or directory) in the current directory, it determines whether it is a file or a directory using `fs.statSync` and `stats.isDirectory()`. It then logs the item's name along with the appropriate label (File or Directory) to the console.

Learning Outcomes :

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