**Practical -2**

**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...");

})

**Description:**

1. const http = require("http");

Node.js has a built-in module called HTTP, which allows Node.js to transfer data over the Hyper Text Transfer Protocol (HTTP). To include the HTTP module, use the require() method.

1. const httpserver = http.createServer(function(req,res){

The HTTP module can create an HTTP server that listens to server ports and gives a response back to the client. Use the createServer() method to create an HTTP server.

1. if(req.method == 'POST')

{

res.end("This is post request");

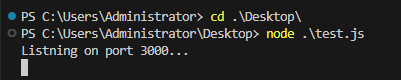
}

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 a post request" using res.end(). The res.end() method ends the response and sends the specified data back to the client.

1. httpserver.listen(3000,()=>{})

The http.server.listen() is an inbuilt application programming interface of the class Server within the HTTP module which is used to start the server by accepting new connections.

**Output:**



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

**Source Code:**

const http = require('http');

const server = http.createServer((req, res) => {

//GET Request

if (req.method === 'GET') {

// Set the response headers

res.writeHead(200, { 'Content-Type': 'text/plain' });

// Write the response message

res.end('Hello, GET request!');

}

//PUT Request

else if (req.method === 'PUT') {

res.writeHead(200, { 'Content-Type': 'text/plain' });

res.end('Hello, PUT request!');

}

//DELETE Request

else if (req.method === 'DELETE') {

res.writeHead(200, { 'Content-Type': 'text/plain' });

res.end('Hello, DELETE request!');

}

// POST Request

else if (req.method === 'POST') {

res.writeHead(200, { 'Content-Type': 'text/plain' });

res.end('Hello, POST request!');

}

// Invalid request method

else {

res.writeHead(400, { 'Content-Type': 'text/plain' });

res.end('Invalid request method');

}

});

// Start the server and listen on port 3000

server.listen(3000, () => {

console.log('Server listening on port 3000');

});

**Description:**

Get Request: The GET method is used to retrieve data from the server. It is a safe and idempotent method, meaning that it should not have any side effects on the server, and making multiple identical GET requests should have the same effect as making a single request.

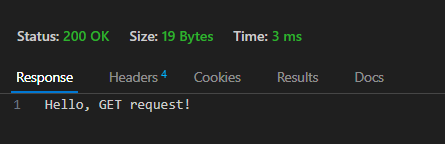
PUT Request: The PUT method is used to update or replace a resource on the server. It is idempotent, meaning that sending the same PUT request multiple times should have the same effect as sending it once. When a client sends a PUT request, it includes the updated representation of the resource in the request payload. The server then processes the payload and updates the resource accordingly.

DELETE Request: The DELETE method is used to request the removal of a resource from the server. When a client sends a DELETE request, it indicates to the server that the resource specified in the request URL should be deleted. The server processes the request and removes the specified resource.

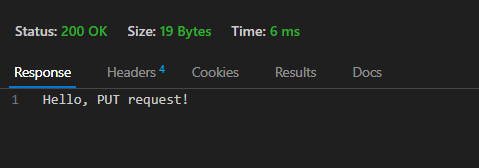
Post Request: The POST method is used to submit data to be processed to a specified resource. It is not idempotent, meaning that making multiple identical POST requests may result in different outcomes on the server each time. When a client sends a POST request, it includes data in the request payload that the server will process and store or perform some action based on the provided data. It is often used for creating new resources on the server.

**Output:**

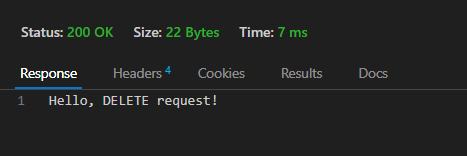
GET Request:



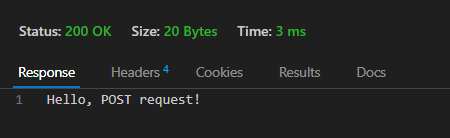
PUT Request:



DELETE Request:



POST Request:



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

**Source Code:**

const fs = require('fs');

const filePath = 'student-data.txt';

// Read the file asynchronously

fs.readFile(filePath, 'utf8', (err, data) => {

if (err) {

console.error('Error reading the file:', err);

return;

}

// Split the data by lines and process each line

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

const filteredStudents = [];

students.forEach(student => {

// Split each line by the comma to get the name and CGPA

const [name, cgpa] = student.split(',');

// Check if the name contains 'MA' and CGPA is greater than 7

if (name.includes('MA') && parseFloat(cgpa) > 7) {

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

}

});

// Output the filtered students

console.log('Filtered Students:');

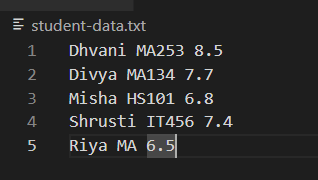
console.log(filteredStudents);

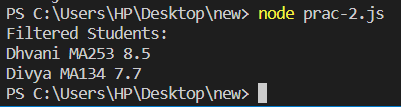
});

**Description:**

* It starts by importing the 'fs' module, which provides functionality to work with the file system.
* The script specifies the file path as 'student-data.txt', where the student data is stored.
* The script reads the file asynchronously using 'fs.readFile()'. If an error occurs during file reading, it will be displayed.
* Once the file is read successfully, the data is split into lines using '\n' as the delimiter.
* Each line representing a student is processed one by one.
* For each student record, the script extracts the student's name and CGPA by splitting the line using a comma as the separator.
* The script checks if the student's name contains 'MA' (case-sensitive) and if the CGPA is greater than 7.
* Students meeting both conditions are added to an array called 'filteredStudents', along with their names and CGPA values.
* After processing all students, the script outputs the list of filtered students to the console.

**Output:**

****

****

**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');

// Create a readline interface to read user input

const rl = readline.createInterface({

input: process.stdin,

output: process.stdout

});

// Function to read employee information from the user

function readEmployeeInfo() {

rl.question('Enter Employee Name: ', (name) => {

rl.question('Enter Employee ID: ', (id) => {

rl.question('Enter Employee Department: ', (department) => {

rl.question('Enter Employee Salary: ', (salary) => {

// Create an object to store employee data

const employee = {

name: name,

id: id,

department: department,

salary: salary

};

// Convert employee object to JSON format

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

// Write the JSON data to the file

fs.writeFile('employee-data.json', jsonData, (err) => {

if (err) {

console.error('Error writing to file:', err);

} else {

console.log('Employee information saved to employee-data.json');

console.log(employee);

}

rl.close();

});

});

});

});

});

}

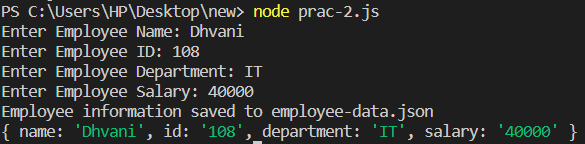
// Start reading employee information

readEmployeeInfo();

**Description:**

* The script begins by importing the necessary modules, readline and fs, which are part of Node.js's standard library.
* It creates a readline interface (rl) using readline.createInterface() to interact with the user through the command-line interface (CLI).
* The function readEmployeeInfo() is defined to read employee information from the user. Inside this function, a series of rl.question() calls are made to prompt the user for various details: employee name, employee ID, employee department, and employee salary.
* The user input for each detail is stored in separate variables (name, id, department, and salary).
* Next, an employee object is created using the user-provided information. This object is then converted to a JSON-formatted string using JSON.stringify().
* The script then writes the JSON data to a file called "employee-data.json" using fs.writeFile(). If the file already exists, this operation will overwrite its content.
* If the file writing operation is successful, the script displays a success message: "Employee information saved to employee-data.json."
* The readline interface is closed using rl.close().
* Finally, the function readEmployeeInfo() is called to start the process of reading and writing employee information.

**Output:**

****

**Aim:** Compare two files and show which file is larger and which lines are different

**Source Code:**

const fs = require('fs');

function compareFiles(file1Path, file2Path) {

try {

// Read the content of the two files

const file1Content = fs.readFileSync(file1Path, 'utf8');

const file2Content = fs.readFileSync(file2Path, 'utf8');

// Compare file sizes

const file1Size = file1Content.length;

const file2Size = file2Content.length;

console.log('Comparison Result:');

if (file1Size > file2Size) {

console.log(`${file1Path} is larger than ${file2Path}.`);

} else if (file1Size < file2Size) {

console.log(`${file2Path} is larger than ${file1Path}.`);

} else {

console.log('Both files have the same size.');

}

// Split the content of the files into lines

const file1Lines = file1Content.split('\n');

const file2Lines = file2Content.split('\n');

// Compare lines and display differences

console.log('\nLines with differences:');

for (let i = 0; i < Math.max(file1Lines.length, file2Lines.length); i++) {

if (file1Lines[i] !== file2Lines[i]) {

console.log(`Line ${i + 1}:`);

console.log(` ${file1Path}: ${file1Lines[i]}`);

console.log(` ${file2Path}: ${file2Lines[i]}`);

console.log('---');

}

}

} catch (err) {

console.error('Error:', err);

}

}

// Compare two files

const file1Path = 'file1.txt';

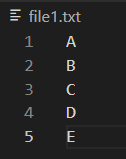
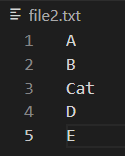
const file2Path = 'file2.txt';

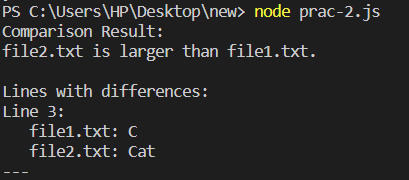
compareFiles(file1Path, file2Path);

**Description:**

* The script starts by importing the fs module, which provides functionality to work with the file system.
* The compareFiles() function is defined to compare two files. It takes the file paths (file1Path and file2Path) as parameters.
* Inside the compareFiles() function, the script attempts to read the content of the two files using fs.readFileSync(). If an error occurs during file reading, it is displayed.
* The file sizes of file1 and file2 are compared to determine which file is larger.
* The result of the file size comparison is displayed using console.log(). It indicates whether file1 is larger, file2 is larger, or if both files have the same size.
* The content of each file is split into lines using the newline character (\n) as the delimiter.
* The script iterates through each line of the files and compares them. If a difference is found, the line number and the corresponding lines from each file are displayed using console.log().
* The script uses the --- separator to visually distinguish between different lines.
* Finally, the script compares two files, specifying the paths of the files (file1Path and file2Path), and calls the compareFiles() function.

**Output:**

**** ****

****

**Aim:** Create File Backup and Restore Utility

**Source Code:**

const fs = require('fs');

const path = require('path');

// Function to create a backup of a file

function createBackup(sourceFilePath, backupFolderPath) {

try {

const sourceFileName = path.basename(sourceFilePath);

const backupFilePath = path.join(backupFolderPath, sourceFileName);

// Read the source file content

const fileContent = fs.readFileSync(sourceFilePath, 'utf8');

// Write the file content to the backup file

fs.writeFileSync(backupFilePath, fileContent, 'utf8');

console.log(`Backup created: ${backupFilePath}`);

} catch (err) {

console.error('Error creating backup:', err);

}

}

// Function to restore a file from a backup

function restoreFromBackup(backupFilePath, destinationFilePath) {

try {

// Read the backup file content

const fileContent = fs.readFileSync(backupFilePath, 'utf8');

// Write the file content to the destination file

fs.writeFileSync(destinationFilePath, fileContent, 'utf8');

console.log(`File restored: ${destinationFilePath}`);

} catch (err) {

console.error('Error restoring file:', err);

}

}

// Example usage:

// Replace 'source-file.txt' with the path of the file you want to backup.

const sourceFilePath = 'file1.txt';

// Replace 'backup-folder' with the folder where you want to store backups.

const backupFolderPath = 'backup-folder';

// Replace 'restore-file.txt' with the path where you want to restore the file.

const restoreFilePath = 'file2.txt';

// Create a backup of the source file

createBackup(sourceFilePath, backupFolderPath);

// Uncomment this line to restore the file from the backup

// restoreFromBackup(path.join(backupFolderPath, path.basename(sourceFilePath)), restoreFilePath);

**Description:**

* The script starts by importing the required modules: fs for file system operations and path for path manipulation.
* Two functions are defined:

createBackup(sourceFilePath, backupFolderPath): This function creates a backup of a specified source file and saves it in the specified backup folder.

restoreFromBackup(backupFilePath, destinationFilePath): This function restores a file from a given backup file and writes it to the specified destination file.

* In the createBackup() function:

It takes the sourceFilePath and backupFolderPath as input parameters, representing the path of the source file to be backed up and the folder where backups will be stored, respectively.

It reads the content of the source file using fs.readFileSync().

It constructs the path for the backup file using path.join() to combine the backupFolderPath with the filename extracted from sourceFilePath.

It writes the content of the source file to the backup file using fs.writeFileSync().

A success message is displayed, confirming that the backup was created.

* In the restoreFromBackup() function:

It takes the backupFilePath and destinationFilePath as input parameters, representing the path of the backup file to be restored and the path where the file should be restored, respectively.

It reads the content of the backup file using fs.readFileSync().

It writes the content of the backup file to the destination file using fs.writeFileSync().

A success message is displayed, confirming that the file has been restored.

* The example usage section demonstrates how to use the utility:

Replace 'source-file.txt' with the path of the file you want to back up.

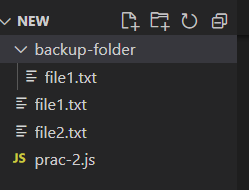
Replace 'backup-folder' with the folder where you want to store backups.

Replace 'restore-file.txt' with the path where you want to restore the file.

* The createBackup() function is called with the provided sourceFilePath and backupFolderPath, creating a backup of the source file.
* The restoreFromBackup() function is commented out to avoid accidental restoration. If you want to restore the file from the backup, you can uncomment this line and provide the appropriate paths.

**Output:**

****

****

**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);

**Description:**

* If – else block

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.

* 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.

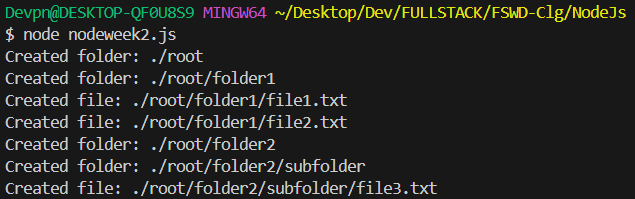
* createFileStructure('.', fileStructure);

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.

**Output:**

****



**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.');

// 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);

}

});

**Description:**

* 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

* 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.

* 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.

* 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.

* fs.unlinkSync('example.txt');

console.log('File deleted.');

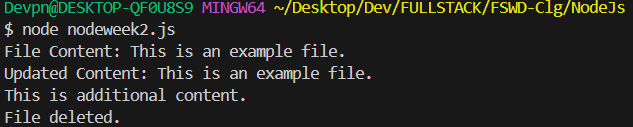
This line deletes the file 'example.txt' using fs.unlinkSync and logs a message indicating that the file has been deleted.

* 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.

**Output:**



**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