**Practical-2**

**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.**

**Theoretical Background:**

* Setting up a basic HTTP server involves creating a Node.js application that utilizes the built-in 'http' module to listen for incoming HTTP requests. By defining a server and handling the 'request' event, we can respond to these requests with a simple message or perform other desired actions.

**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 is running on http://localhost:3000');

});



**Output:**

**Task-2**

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

**Theoretical Background:**

* An experiment involving various HTTP methods, content types, and status codes explores the interactions between clients and servers in web communication. It aims to analyze how different methods (such as GET, POST, PUT, DELETE) and content types (such as JSON, XML, HTML) impact the response status codes, providing insights into effective data exchange and error handling in web development.

**Source Code:**

const http = require('http');

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

const { method, url } = req;

if (method === 'GET') {

if (url === '/') {

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

res.end('This is a GET request');

} else {

res.writeHead(404, { 'Content-Type': 'text/plain' });

res.end('Not Found');

}

} else if (method === 'POST') {

if (url === '/data') {

res.writeHead(201, { 'Content-Type': 'text/plain' });

res.end('Data created successfully');

} else {

res.writeHead(404, { 'Content-Type': 'text/plain' });

res.end('Not Found');

}

} else {

res.writeHead(405, { 'Content-Type': 'text/plain' });

res.end('Method Not Allowed');

}

});

server.listen(3000, () => {

console.log('Server is running on http://localhost:3000');

});

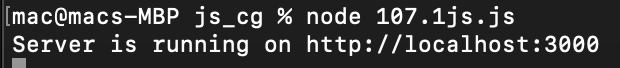


**Output:**

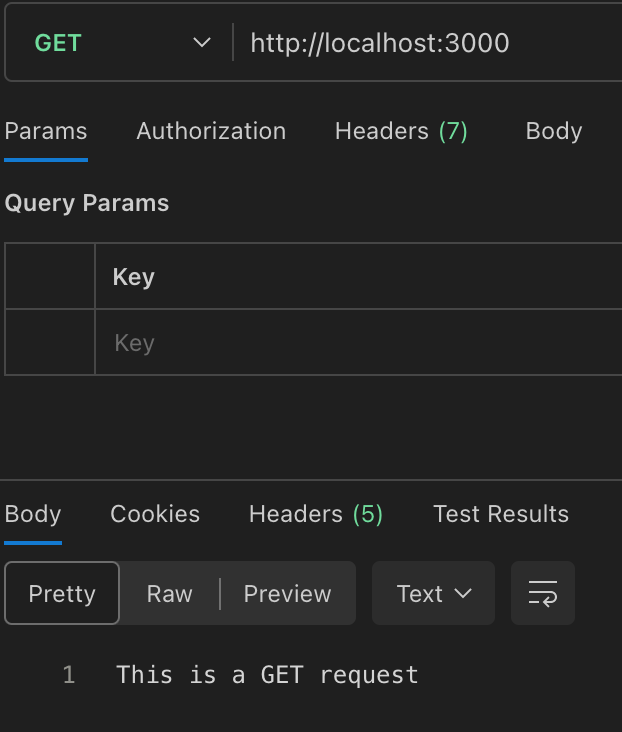
**Task-3**

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

**Theoretical Background:**

* Testing an application using a browser involves simulating user interactions and verifying the expected behavior through manual or automated techniques within the web environment.
* CLI testing involves executing commands or scripts via the command line interface to validate the functionality, inputs, and outputs of a software application.
* Testing a RESTful API using a REST client involves sending HTTP requests and verifying the responses to ensure the correct implementation of the API endpoints and their associated functionality.

**Source Code:**

**Output:**



**Task-4**

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

**Theoretical Background:**

* reading a file named "student-data.txt" and finding all students whose name contains 'MA' and have a CGPA greater than 7 involves opening the file, reading its contents line by line, splitting each line into relevant fields (such as name and CGPA), and applying filtering conditions to identify the desired students based on their name and CGPA values.

**Source Code:**

const fs = require('fs');

const students = [];

fs.readFile('student-data.txt', 'utf8', (err, data) => {

if (err) {

console.error(err);

return;

}

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

for (const line of lines) {

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

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

students.push({ name, cgpa });

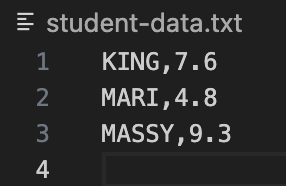
}

}

console.log(students);

});

**Output:**

****

**Task-5**

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

**Theoretical Background:**

* The theoretical background for reading employee information from a user and writing the data to a file called 'employee-data.json' involves utilizing programming concepts such as user input and file I/O operations to create an interactive system that collects and stores employee details in a structured JSON format.

**Source Code:**

<!DOCTYPE html>

<html>

<head>

<title>Employee Information</title>

<style>

label {

display: block;

margin-top: 10px;

}

</style>

</head>

<body>

<h1>Employee Information</h1>

<form id="employee-form">

<label for="name">Name:</label>

<input type="text" id="name" required>

<label for="age">Age:</label>

<input type="number" id="age" required>

<label for="salary">Salary:</label>

<input type="number" id="salary" required>

<button type="submit">Submit</button>

</form>

<script>

document.getElementById('employee-form').addEventListener('submit', function(event) {

event.preventDefault();

const name = document.getElementById('name').value;

const age = document.getElementById('age').value;

const salary = document.getElementById('salary').value;

const employee = {

name: name,

age: age,

salary: salary

};

const employeeJSON = JSON.stringify(employee);

const blob = new Blob([employeeJSON], { type: 'application/json' });

const link = document.createElement('a');

link.href = URL.createObjectURL(blob);

link.download = 'employee-data.json';

document.body.appendChild(link);

link.click();

document.body.removeChild(link);

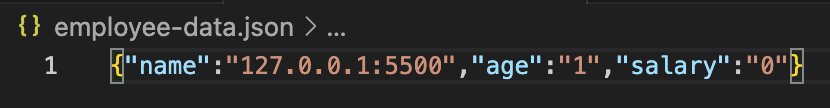
URL.revokeObjectURL(link.href);

});

</script>

</body>

</html>



**Output:**

**Task-6**

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

**Theoretical Background:**

* Comparing two files involves analyzing their content to determine the file size difference and identifying the lines that are different between them. This process is commonly used in data analysis, version control systems, and file comparison tools to detect variations and facilitate decision-making based on discrepancies.

**Source Code:**

const fs = require('fs');

const file1 = 'file1.txt';

const file2 = 'file2.txt';

const data1 = fs.readFileSync(file1, 'utf8');

const data2 = fs.readFileSync(file2, 'utf8');

const lines1 = data1.split('\n');

const lines2 = data2.split('\n');

const largerFile = lines1.length > lines2.length ? file1 : file2;

const smallerFile = lines1.length < lines2.length ? file1 : file2;

const differentLines = [];

for (let i = 0; i < lines1.length; i++) {

if (lines1[i] !== lines2[i]) {

differentLines.push(`${i + 1}: ${lines1[i]} ≠ ${lines2[i]}`);

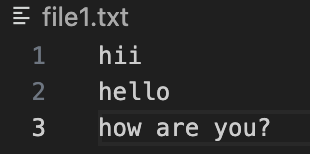
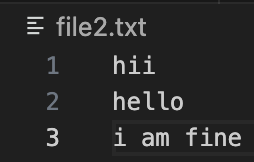
}

}

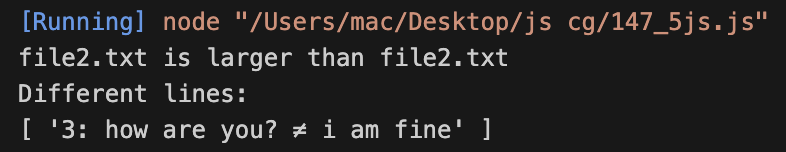
console.log(`${largerFile} is larger than ${smallerFile}`);

console.log('Different lines:');

console.log(differentLines);



**Output:**



**Task-7**

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

**Theoretical Background:**

* A Create File Backup and Restore Utility in JavaScript allows users to safeguard their data by creating copies of files and restoring them when needed, ensuring data integrity and minimizing the risk of data loss.

**Source Code:**

const fs = require('fs');

const path = require('path');

const { exec } = require('child\_process');

function createBackup(filePath) {

const backupPath = filePath + '.bak';

fs.copyFileSync(filePath, backupPath);

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

}

function restoreBackup(backupPath) {

const filePath = backupPath.slice(0, -4);

fs.copyFileSync(backupPath, filePath);

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

}

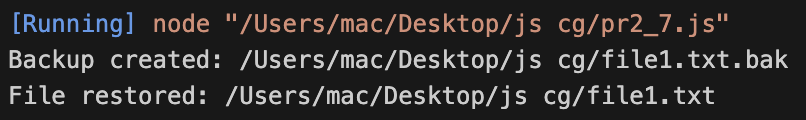
const fileToBackup = path.join(\_\_dirname, 'file1.txt');

createBackup(fileToBackup);

fs.writeFileSync(fileToBackup, 'Modified content');

const backupFile = fileToBackup + '.bak';

restoreBackup(backupFile);



**Output:**

**Task-8**

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

**Theoretical Background:**

* Creating file/folder structures from a JSON file involves parsing the JSON data and using the information provided to programmatically generate the desired file and folder hierarchy.

**Source Code:**

const fs = require('fs');

const jsonData = {

"root": {

"folder1": {

"file1.txt": "This is file 1",

"file2.txt": "This is file 2"

},

"folder2": {

"subfolder1": {

"file3.txt": "This is file 3"

},

"file4.txt": "This is file 4"

}

}

};

function createStructure(parentPath, data) {

for (const key in data) {

const currentPath = `${parentPath}/${key}`;

const value = data[key];

if (typeof value === 'object') {

fs.mkdirSync(currentPath);

createStructure(currentPath, value);

} else {

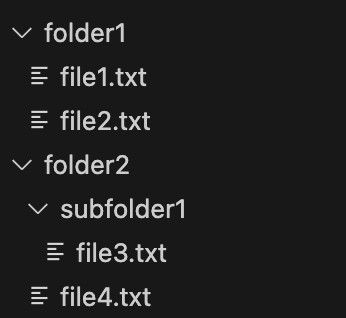
fs.writeFileSync(currentPath, value);

}

}

}

createStructure('.', jsonData.root);

**Output:**

**Task-9**

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

**Theoretical Background:**

* Experimenting with file operations such as creating, reading, appending, deleting, renaming files, and listing files/directories helps understand fundamental aspects of file management and data manipulation within a computer system.

**Source Code:**

const fs = require('fs');

const path = require('path');

fs.writeFile('example.txt', 'Hello, World!', (err) => {

if (err) throw err;

console.log('File created successfully!');

});

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

if (err) throw err;

console.log('File content:', data);

});

fs.appendFile('example.txt', '\nAppended content!', (err) => {

if (err) throw err;

console.log('Content appended successfully!');

});

fs.rename('example.txt', 'new-example.txt', (err) => {

if (err) throw err;

console.log('File renamed successfully!');

});

fs.unlink('new-example.txt', (err) => {

if (err) throw err;

console.log('File deleted successfully!');

});

const directoryPath = './';

fs.readdir(directoryPath, (err, files) => {

if (err) throw err;

console.log('Files and directories in', directoryPath);

files.forEach((file) => {

console.log(file);

});

});

**Output:**

File deleted successfully!

File renamed successfully!

Files and directories in ./

file1.txt

file1.txt.bak

file2.txt

folder1

folder2

pr2\_9.js

pr3\_2.mjs

File created successfully!

Content appended successfully!

File content: Hello, World!tent!

**Learning Outcome:**

* Understanding javascript technologies(CO1).