Experiment No: 8

Aim : Study the installation of Node JS and installation of various packages in NodeJS.

Quiz:

1. Write a note on : Routing in Angular JS.

   Routing in AngularJS is a crucial aspect of building single-page applications (SPAs). It allows developers to define different views and map them to specific URLs within the application. This enables users to navigate between different parts of the application seamlessly without the need for page reloads.

Key Concepts:

Routes: Routes in AngularJS are defined using the $routeProvider service. Developers can specify the URL, the corresponding template to render, and the controller associated with that template.

Single Page Application (SPA): AngularJS uses client-side routing to render different views without requesting new pages from the server. This approach enhances user experience by providing a faster and more responsive application.

Route Parameters: AngularJS allows developers to define dynamic routes with parameters. These parameters can be passed in the URL and accessed within the controller to fetch specific data or perform actions.

Route Events: AngularJS provides several route-related events such as $routeChangeStart, $routeChangeSuccess, and $routeChangeError. These events allow developers to execute code before or after a route change, enabling tasks such as authentication checks or loading indicators.

2. State the usefulness of ng-view directive. Explain various ways to use it.

   The ng-view directive in AngularJS serves as a placeholder where AngularJS inserts views based on the current route. It plays a crucial role in implementing client-side routing and dynamically loading content without page refresh. Here's why it's useful and how it can be utilized:

Usefulness:

Dynamic Content Loading: ng-view allows developers to load different templates and controllers based on the current route, providing a dynamic and interactive user experience without the need for full page reloads.

Route-Driven Architecture: By defining routes and associating them with specific templates, controllers, and views, developers can create a route-driven architecture that enhances code organization and maintainability.

Single Page Application (SPA) Support: ng-view facilitates the development of SPAs by enabling client-side routing, where different views can be rendered within the same HTML page based on user interactions or URL changes.

SEO Optimization: While SPAs primarily render content on the client-side, techniques such as server-side rendering can be combined with ng-view to improve search engine indexing and SEO.

Assessment :

Node JS installation process (4 marks)	Understanding of NPM (3 marks)	Presentation and report writing (3 marks)	Total (10 marks)

Experiment No: 9

Aim : Design a webpage with a file input control to browse appropriate file and four buttons Read File Synchronously, Read File Asynchronously, Compress File, Decompress File. Implement the functionality of all four buttons on the browsed file using Node JS.

App.js

```
const http = require("http");
const fs = require("fs");
const formidable = require("formidable");
const zlib = require("zlib");

const server = http.createServer((req, res) => {
  if (req.url === "/") {
    fs.readFile("index.html", (err, data) => {
      if (err) {
        res.writeHead(500);
        res.end(err.message);
      } else {
        res.writeHead(200, { "Content-Type": "text/html" });
        res.end(data);
      }
    });
  } else if (req.url === "/readFileSync" && req.method === "POST") {
    const form = new formidable.IncomingForm();
    form.uploadDir = "./uploads";
    form.keepExtensions = true;
    form.parse(req, (err, fields, files) => {
      if (err) {
        res.writeHead(500);
        res.end(err.message);
      } else {
        const uploadedFile = files.file[0];
        const filePath = uploadedFile.filepath;
        const fileContent = fs.readFileSync(filePath, "utf8");
        res.writeHead(200, { "Content-Type": "text/plain" });
        res.end(fileContent);
      }
    });
  } else if (req.url === "/readFileAsync" && req.method === "POST") {
    const form = new formidable.IncomingForm();
    form.uploadDir = "./uploads";
    form.keepExtensions = true;
    form.parse(req, (err, fields, files) => {
      if (err) {
        res.writeHead(500);
        res.end(err.message);
      } else {
        const uploadedFile = files.file[0];
        const filePath = uploadedFile.filepath;
        fs.readFile(filePath, "utf8", (err, fileContent) => {
```

```
        if (err) {
          res.writeHead(500);
          res.end(err.message);
        } else {
          res.writeHead(200, { "Content-Type": "text/plain" });
          res.end(fileContent);
        }
      });
    }
  });
}

else if (req.url === "/compressFile" && req.method === "POST") {
  const form = new formidable.IncomingForm();
  form.uploadDir = "./uploads";
  form.keepExtensions = true;
  form.parse(req, (err, fields, files) => {
    if (err) {
      res.writeHead(500);
      res.end(err.message);
    } else {
      const uploadedFile = files.file[0];
      const filePath = uploadedFile.filepath;

      const readStream = fs.createReadStream(filePath);

      const compressedFilePath = filePath + '.gz';
      const writeStream = fs.createWriteStream(compressedFilePath);

      readStream.pipe(zlib.createGzip()).pipe(writeStream);

      writeStream.on('finish', () => {
        res.setHeader('Content-disposition', 'attachment; filename=' +
uploadedFile.originalFilename + '.gz');
        res.setHeader('Content-Type', 'application/gzip');

        const compressedReadStream = fs.createReadStream(compressedFilePath);
        compressedReadStream.pipe(res);
      });
      writeStream.on('error', (error) => {
        res.writeHead(500);
        res.end(error.message);
      });
    }
  });
}

else if (req.url === "/decompressFile" && req.method === "POST") {
  const form = new formidable.IncomingForm();
  form.uploadDir = "./uploads";
  form.keepExtensions = true;
  form.parse(req, (err, fields, files) => {
    if (err) {
      res.writeHead(500);
```

```
        res.end(err.message);
      } else {
        const uploadedFile = files.file[0];
        const filePath = uploadedFile.filepath;

        // Decompressing the file
        const readStream = fs.createReadStream(filePath);
        const unzipStream = zlib.createGunzip();
        const writeStream =
fs.createWriteStream(`./uploads/${uploadedFile.originalFilename.substring(0,
uploadedFile.originalFilename.length - 3)}`);

        readStream.pipe(unzipStream).pipe(writeStream);

        writeStream.on('finish', () => {
          res.writeHead(200, { "Content-Type": "text/plain" });
          res.end("File decompressed successfully");
        });

        writeStream.on('error', (error) => {
          res.writeHead(500);
          res.end(error.message);
        });
      }
    });
  }
});

server.listen(5000, () => {
  console.log("Server is running on port 5000");
});
```

Index.html

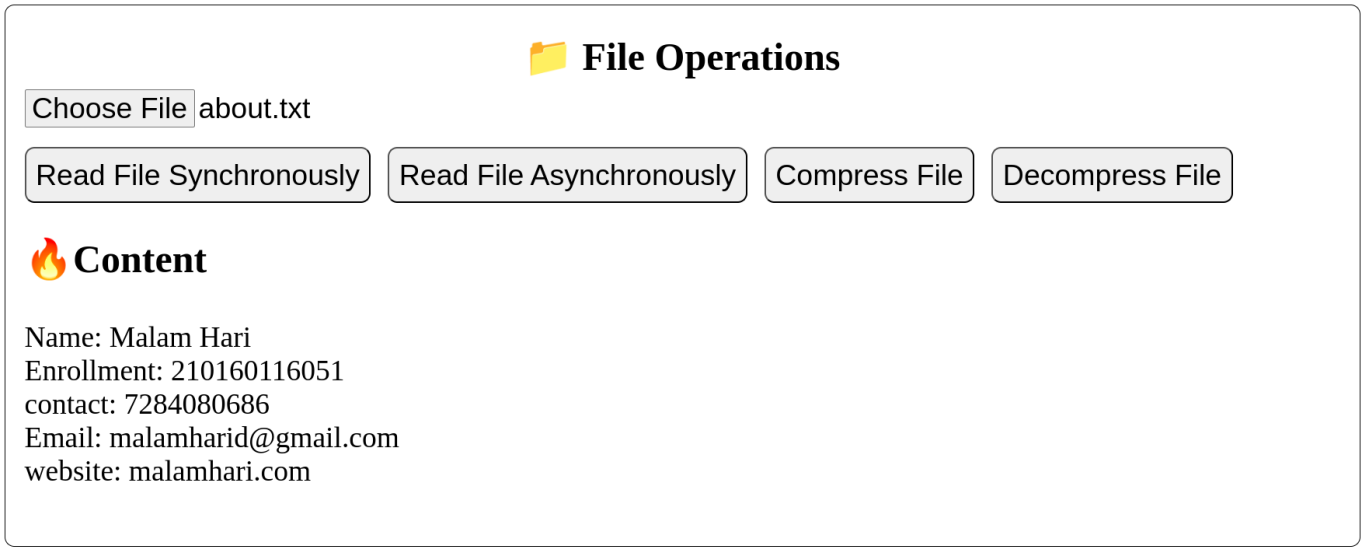
```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>File Operations</title>
    <style>
      * {
        padding: 0;
        margin: 0;
        font-size: 30px;
      }
      button {
        padding: 10px;
        margin-right: 10px;
        margin-top: 20px;
        border-radius: 10px;
      }
      .input {
        padding: 20px;
```

```
      margin: 30px 70px;
      border: 1px solid black;
      border-radius: 10px;
    }
    h1 {
      margin: 10px 0px;
      text-align: center;
      font-size: 40px;
    }
    h2 {
      font-size: 40px;
    }
    #fileContent {
      padding: 40px 0px;
    }
  </style>
</head>
<body>
  <div class="input">
    <h1>📁 File Operations</h1>
    <input type="file" id="fileInput" />
    <br />
    <button onclick="readFileSync()">Read File Synchronously</button>
    <button onclick="readFileAsync()">Read File Asynchronously</button>
    <button onclick="compressFile()">Compress File</button>
    <button onclick="decompressFile()">Decompress File</button>
    <br />
    <br />
    <h2>🔥Content</h2>
    <div id="fileContent"></div>
  </div>

  <script>
    function readFileSync() {
      const fileInput = document.getElementById("fileInput");
      const file = fileInput.files[0];
      if (file) {
        const formData = new FormData();
        formData.append("file", file);

        fetch("/readFileSync", {
          method: "POST",
          body: formData,
        })
          .then((response) => response.text())
          .then((data) => {
            document.getElementById("fileContent").innerText = data;
          })
          .catch((error) => console.error("Error:", error));
      }
    }

    function readFileAsync() {
      const fileInput = document.getElementById("fileInput");
```

```
  const file = fileInput.files[0];
  if (file) {
    const formData = new FormData();
    formData.append("file", file);

    fetch("/readFileAsync", {
      method: "POST",
      body: formData,
    })
      .then((response) => response.text())
      .then((data) => {
        document.getElementById("fileContent").innerText = data;
      })
      .catch((error) => console.error("Error:", error));
  }
}

function compressFile() {
  const fileInput = document.getElementById("fileInput");
  const file = fileInput.files[0];
  if (file) {
    const formData = new FormData();
    formData.append("file", file);

    fetch("/compressFile", {
      method: "POST",
      body: formData,
    })
      .then((response) => {
        if (response.ok) {
          // Trigger file download
          response.blob().then((blob) => {
            const url = window.URL.createObjectURL(blob);
            const a = document.createElement("a");
            a.href = url;
            a.download = `${file.name}.gz`;
            document.body.appendChild(a);
            a.click();
            window.URL.revokeObjectURL(url);
          });
        } else {
          throw new Error("Compression failed");
        }
      })
      .catch((error) => console.error("Error:", error));
  }
}

function decompressFile() {
  const fileInput = document.getElementById("fileInput");
  const file = fileInput.files[0];
  if (file) {
    const formData = new FormData();
    formData.append("file", file);
```

```
        fetch("/decompressFile", {
          method: "POST",
          body: formData,
        })
          .then((response) => {
            if (response.ok) {
              // Trigger file download
              response.blob().then((blob) => {
                const url = window.URL.createObjectURL(blob);
                const a = document.createElement("a");
                a.href = url;
                a.download = `${file.name.substring(
                  0,
                  file.name.length - 3
                )}`; // Remove the .gz extension
                document.body.appendChild(a);
                a.click();
                window.URL.revokeObjectURL(url);
              });
            } else {
              throw new Error("Decompression failed");
            }
          })
          .catch((error) => console.error("Error:", error));
      }
    }
  </script>
 </body>
</html>
```

Output



Quiz:

1. Enlist various functions available in **fs** module and also state their usages.

   fs.readFile(path[, options], callback): Reads the contents of a file asynchronously. It takes the file path, options (like encoding), and a callback function that gets called with the error (if any) and the file data.

   fs.readFileSync(path[, options]): Synchronous version of fs.readFile(). It returns the file data directly instead of using a callback.

   fs.writeFile(file, data[, options], callback): Asynchronously writes data to a file, replacing the file if it already exists. It takes the file path, data to be written, options (like encoding), and a callback function to handle errors.

   fs.writeFileSync(file, data[, options]): Synchronous version of fs.writeFile(). It writes data to a file and returns undefined.

   fs.readdir(path[, options], callback): Reads the contents of a directory asynchronously. It takes the directory path, options (like encoding), and a callback function that gets called with the error (if any) and an array of directory entries.

   fs.readdirSync(path[, options]): Synchronous version of fs.readdir(). It returns an array of directory entries directly instead of using a callback.

   fs.unlink(path, callback): Asynchronously removes a file or symbolic link. It takes the file path and a callback function to handle errors.

   fs.unlinkSync(path): Synchronous version of fs.unlink(). It removes a file or symbolic link and returns undefined.

   fs.stat(path, callback): Asynchronously gets the file status. It takes the file path and a callback function that gets called with the error (if any) and a Stats object representing the file status.

   fs.statSync(path): Synchronous version of fs.stat(). It returns a Stats object representing the file status directly instead of using a callback.

2. Explain various functions available in **zlib** module.

   zlib.gzip(input, options, callback): Compresses a buffer or string asynchronously using gzip compression. It takes the input data, options (like compression level), and a callback function that gets called with the error (if any) and the compressed data.

   zlib.gzipSync(input, options): Synchronous version of zlib.gzip(). It compresses data using gzip compression and returns the compressed data directly instead of using a callback.

   zlib.gunzip(buffer, options, callback): Decompresses a buffer asynchronously using gzip compression. It takes the compressed data buffer, options (like output encoding), and a callback function that gets called with the error (if any) and the decompressed data.

   zlib.gunzipSync(buffer, options): Synchronous version of zlib.gunzip(). It decompresses data using gzip compression and returns the decompressed data directly instead of using a callback.

   zlib.deflate(input, options, callback): Compresses a buffer or string asynchronously using the deflate algorithm. It takes the input data, options (like compression level), and a callback function that gets called with the error (if any) and the compressed data.

   zlib.deflateSync(input, options): Synchronous version of zlib.deflate(). It compresses data using the deflate algorithm and returns the compressed data directly instead of using a

callback.

zlib.inflate(buffer, options, callback): Decompresses a buffer asynchronously using the deflate algorithm. It takes the compressed data buffer, options (like output encoding), and a callback function that gets called with the error (if any) and the decompressed data.

zlib.inflateSync(buffer, options): Synchronous version of zlib.inflate(). It decompresses data using the deflate algorithm and returns the decompressed data directly instead of using a callback.

Assessment :

Understanding of Problem (3 marks)	Implementation of Problem (4 marks)	Presentation and report writing (3 marks)	Total (10 marks)

Experiment No: 10

Aim : Create a Node JS application that will allow a user to browse and upload a file in localhost.

App.js

```
const http = require("http");
const fs = require("fs");
const formidable = require("formidable");

http
  .createServer(function (req, res) {
    // Handle file upload request
    if (req.url == "/upload") {
      var form = new formidable.IncomingForm();

      form.parse(req, function (err, fields, files) {
        // Get the old path of the uploaded file
        var oldpath = files.file[0].filepath;
        console.log(oldpath);

        var newpath = __dirname + "/uploads/" + files.file[0].originalFilename;

        // Move the uploaded file to the new path
        const readStream = fs.createReadStream(oldpath);
        const writeStream = fs.createWriteStream(newpath);
        readStream.pipe(writeStream);
        writeStream.on("error", function (err) {
          console.error("Error copying file:", err);
          res.writeHead(500, { "Content-Type": "text/plain" });
          res.end("Error uploading file");
        });

        // When copying is complete, respond to the client
        writeStream.on("finish", function () {
          res.writeHead(200, { "Content-Type": "text/plain" });
          res.end("File uploaded successfully");
        });
      });
    } else {
      fs.readFile("index.html", (err, data) => {
        if (err) {
          res.writeHead(500);
          res.end(err.message);
        } else {
          res.writeHead(200, { "Content-Type": "text/html" });
          res.end(data);
        }
      });
    }
  })
  .listen(300);
```

Index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>File Upload</title>
  <style>
    body{
      display: flex;
      justify-content: center;
    }
    *{
      font-size: 35px;
    }
    h1{
      font-size: 60px;
      margin-bottom: 50px;
    }
    .container{
      text-align: center;
      border: 1px solid black;
      padding: 30px;
      border-radius: 10px;
    }
  </style>
</head>
<body>
  <div class="container">
    <h1>📁 Upload a File</h1>
    <form action="/upload" method="post" enctype="multipart/form-data">
      <input type="file" name="file">
      <button type="submit">Upload</button>
    </form>
  </div>
</body>
</html>
```

Output:

Quiz:

1. Enlist and explain various methods of formidable module used to manage uploading files.
   1. Formidable Constructor:
   new formidable.IncomingForm(): Creates a new instance of the IncomingForm class, which is used to parse incoming form data, including file uploads.

   2. Form Parsing Methods:
   form.parse(req, callback): Parses the incoming form data from the request object. It extracts fields and files from the form and invokes the provided callback with any encountered errors and the parsed fields and files.

   3. Event Handlers:
   form.on('file', function(field, file): Event handler triggered for each file field encountered during form parsing. It provides the field name and file object.
   form.on('field', function(name, value): Event handler triggered for each non-file field encountered during form parsing. It provides the field name and its value.
   form.on('error', function(err): Event handler triggered when an error occurs during form parsing.

   4. File Management Methods:
   file.uploadDate: Provides the date when the file was uploaded.
   file.path: Gives the temporary path of the uploaded file on the server.
   file.name: Provides the name of the uploaded file.
   file.type: Gives the MIME type of the uploaded file.
   file.size: Provides the size of the uploaded file in bytes.

   5. Configuration Options:
   form.encoding: Specifies the encoding for the incoming form data.
   form.uploadDir: Sets the directory where uploaded files will be stored temporarily.
   form.keepExtensions: Indicates whether to keep file extensions when saving uploaded files.
   form.maxFieldsSize: Specifies the maximum size (in bytes) of all fields together.

   6. Parsing File Uploads:
   form.parse(req, function(err, fields, files): Parses the incoming form data from the request object and invokes the provided callback with any encountered errors, parsed fields, and uploaded files.

Assessment :

Understanding of Problem (3 marks)	Implementation of Problem (4 marks)	Presentation and report writing (3 marks)	Total (10 marks)

Experiment No: 11

Aim : Create a Node JS application that will allow a user to create new file, read file, write into a file and delete a file.

App.js

```
const express = require("express");
const fs = require("fs");
const bodyParser = require("body-parser");
const path = require("path");

const app = express();
const PORT = 3060;

// Middleware for parsing JSON body
app.use(bodyParser.json());
app.use(express.urlencoded({ extended: false }));


const mypath = path.join(__dirname + "/views");

app.get("/", (req, res) => {
  res.sendFile(`${mypath}/index.html`);
});


// Route to create a new file
app.post("/create", (req, res) => {
  const { createFileName } = req.body;
  const filepath = path.join(__dirname + "/files/" + createFileName);
  console.log(filepath);
  fs.open(filepath, "w", (err, fd) => {
    if (err) {
      res.status(500).json({ error: err.message });
      return;
    }
    fs.close(fd, (err) => {
      if (err) console.error(err);
      // res.json({ message: `${createFileName} created successfully.` });
      res.redirect('/');
    });
  });
});

// Route to read a file
app.post("/read", (req, res) => {
  const { readFileName } = req.body;
  const filepath = path.join(__dirname + "/files/" + readFileName);
  fs.readFile(filepath, "utf8", (err, data) => {
    if (err) {
      res.status(404).json({ error: `File ${readFileName} not found.` });
      return;
    }
```

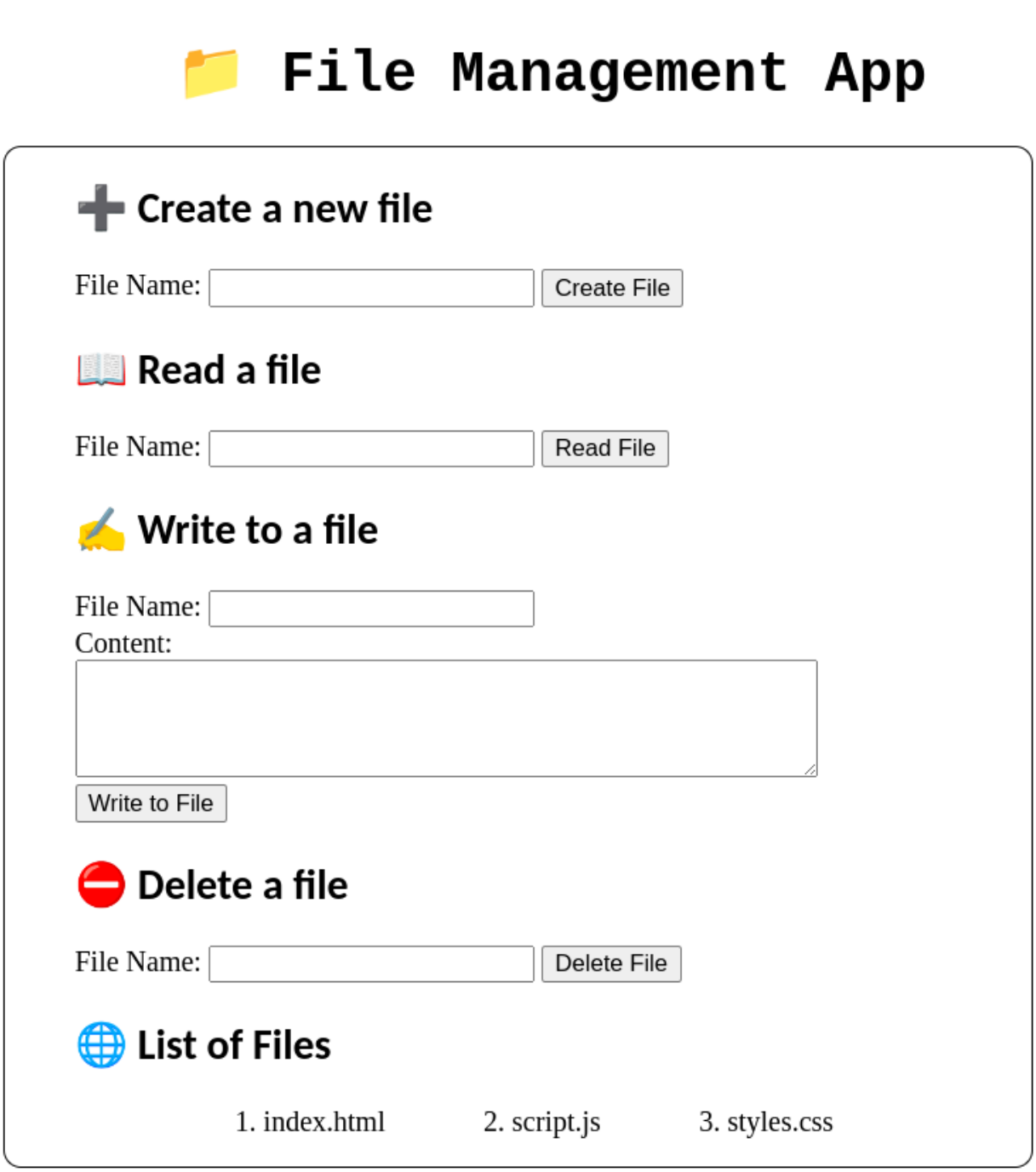
```
    res.json({ content: data });
  });
});

// Route to write to a file
app.post("/write", (req, res) => {
  const { writeFileName, writeContent } = req.body;
  const filepath = path.join(__dirname + "/files/" + writeFileName);
  fs.writeFile(filepath, writeContent, (err) => {
    if (err) {
      res.status(500).json({ error: err.message });
      return;
    }
    res.json({ message: `${writeFileName} updated successfully.` });
  });
});

// Route to delete a file
app.post("/delete", (req, res) => {
  const { deleteFileName } = req.body;
  const filepath = path.join(__dirname + "/files/" + deleteFileName);
  fs.unlink(filepath, (err) => {
    if (err) {
      res.status(500).json({ error: err.message });
      return;
    }
    // res.json({ message: `${deleteFileName} deleted successfully.` });
    res.redirect('/');
  });
});

// Route to fetch list of files
app.get("/files", (req, res) => {
  const filepath = path.join(__dirname + "/files/");
  fs.readdir(filepath, (err, files) => {
    if (err) {
      res.status(500).json({ error: err.message });
      return;
    }
    res.json(files);
  });
});

// Start the server
app.listen(PORT, () => {
  console.log(`Server is running on http://localhost:${PORT}`);
});
```

Index.html

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>File Management App</title>
    <style>
      h1 {
        text-align: center;
        font-family: "Courier New", Courier, monospace;
      }
      .action {
        border: 1px solid black;
        padding: 0px 40px;
        align-items: center;
        border-radius: 10px;
        width: 500px;
        margin-left: 450px;
      }
      ol{
        display: flex;
        flex-wrap: wrap;
        justify-content: space-evenly;
      }
      li{
        padding-right: 5px;
      }
      h2{
        font-family: 'Gill Sans', 'Gill Sans MT', Calibri, 'Trebuchet MS', sans-serif;
      }
    </style>
  </head>
  <body>
    <h1>📁 File Management App</h1>
      <div class="action">
        <!-- Create File Form -->
        <h2>➕ Create a new file</h2>
        <form id="createForm" method="post" action="/create">
          <label for="createFileName">File Name:</label>
          <input type="text" id="createFileName" name="createFileName" />
          <button type="submit">Create File</button>
        </form>

        <!-- Read File Form -->
        <h2>📖 Read a file</h2>
        <form id="readForm" method="post" action="/read">
          <label for="readFileName">File Name:</label>
          <input type="text" id="readFileName" name="readFileName" />
          <button type="submit">Read File</button>
        </form>

        <!-- Write to File Form -->
        <h2>✍️ Write to a file</h2>
```

```
        <form id="writeForm" method="post" action="/write">
          <label for="writeFileName">File Name:</label>
          <input type="text" id="writeFileName" name="writeFileName" /><br />
          <label for="writeContent">Content:</label><br />
          <textarea
            id="writeContent"
            name="writeContent"
            rows="4"
            cols="50"
          ></textarea
          ><br />
          <button type="submit">Write to File</button>
        </form>

        <!-- Delete File Form -->
        <h2>⛔ Delete a file</h2>
        <form id="deleteForm" method="post" action="delete">
          <label for="deleteFileName">File Name:</label>
          <input type="text" id="deleteFileName" name="deleteFileName" />
          <button type="submit">Delete File</button>
        </form>
        <div class="list">
          <!-- List of Files -->
          <h2>🌐 List of Files</h2>
          <ol id="fileList"></ol>
        </div>
      </div>

    <script>
      function fetchFileList() {
        fetch("/files")
          .then((response) => response.json())
          .then((data) => {
            const fileListElement = document.getElementById("fileList");
            fileListElement.innerHTML = ""; // Clear previous list
            data.forEach((fileName) => {
              const listItem = document.createElement("li");
              listItem.textContent = fileName;
              fileListElement.appendChild(listItem);
            });
          })
          .catch((error) => console.error("Error:", error));
      }

      // Fetch file list initially when the page loads
      window.addEventListener("DOMContentLoaded", () => {
        fetchFileList();
      });
    </script>
  </body>
</html>
```

Output:



Practical Quiz:

1. Create Node JS application that will list all the files available in the browsed directory of a server.

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

const fs = require('fs');

const path = require('path');


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

```
    const directoryPath = '.'; // Change this to the desired directory
    fs.readdir(directoryPath, (err, files) => {
        if (err) {
            res.writeHead(500, { 'Content-Type': 'text/plain' });
            res.end('Internal Server Error');
            return;
        }
        res.writeHead(200, { 'Content-Type': 'text/html' });
        res.write('<h1>Files in Directory</h1>');
        res.write('<ul>');
        files.forEach(file => {
            res.write(`<li>${file}</li>`);
        });
        res.write('</ul>');
        res.end();
    });
});

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

2. Create Node JS application that will allow a user to create new files and rename existing files using the proper interface

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

const server = http.createServer((req, res) => {
    const parsedUrl = url.parse(req.url, true);
    const pathname = parsedUrl.pathname;
```

```
    if (pathname === '/create') {
        const filename = parsedUrl.query.filename;
        fs.writeFile(filename, 'Content of the new file', (err) => {
            if (err) {
                res.writeHead(500, { 'Content-Type': 'text/plain' });
                res.end('Internal Server Error');
                return;
            }
            res.writeHead(200, { 'Content-Type': 'text/plain' });
            res.end(`File ${filename} created successfully.`);
        });
    } else if (pathname === '/rename') {
        const oldName = parsedUrl.query.oldName;
        const newName = parsedUrl.query.newName;
        fs.rename(oldName, newName, (err) => {
            if (err) {
                res.writeHead(500, { 'Content-Type': 'text/plain' });
                res.end('Internal Server Error');
                return;
            }
            res.writeHead(200, { 'Content-Type': 'text/plain' });
            res.end(`File ${oldName} renamed to ${newName} successfully.`);
        });
    } else {
        res.writeHead(404, { 'Content-Type': 'text/plain' });
        res.end('Page not found');
    }
});

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

Assessment :

Understanding of Problem (3 marks)	Implementation of Problem (4 marks)	Presentation and report writing (3 marks)	Total (10 marks)

Experiment No: 12

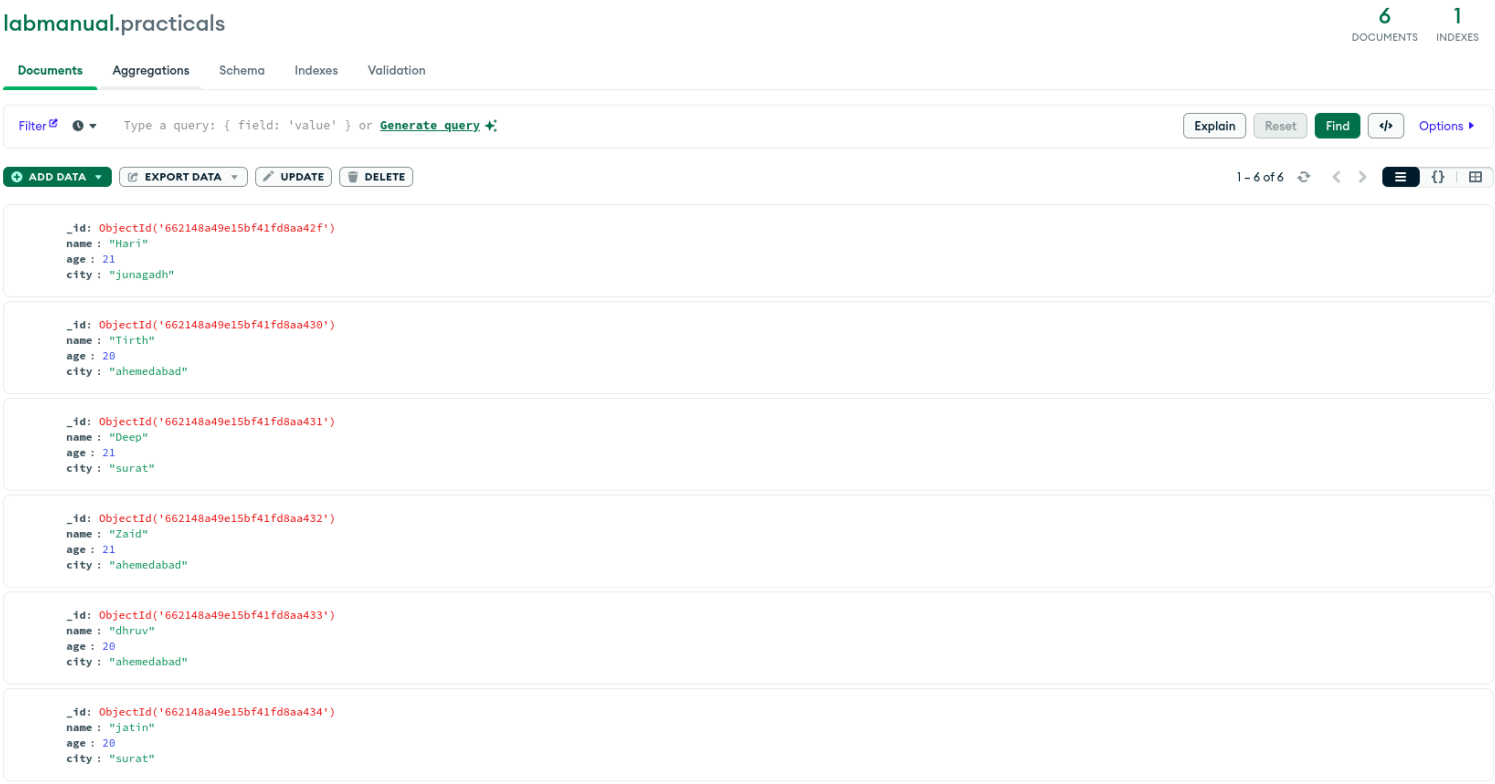
Aim : Study MongoDB environment setup and write Node JS code to perform insertion operation in Mongo DB.

App.js

```
const { MongoClient } = require("mongodb");
const uri = "mongodb://127.0.0.1:27017";
const dbName = "labmanual";
const collectionName = "practicals";

const data = [
  { name: "Hari", age: 21, city: "junagadh" },
  { name: "Tirth", age: 20, city: "ahemedabad" },
  { name: "Deep", age: 21, city: "surat" },
  { name: "Zaid", age: 21, city: "ahemedabad" },
  { name: "dhruv", age: 20, city: "ahemedabad" },
  { name: "jatin", age: 20, city: "surat" },
];

async function insertData() {
  const client = new MongoClient(uri);
  try {
    await client.connect();
    const db = client.db(dbName);
    const collection = db.collection(collectionName);

    const result = await collection.insertMany(data);

    console.log(`${result.insertedCount} documents inserted.`);
  } catch (error) {
    console.error("Error inserting data:", error);
  } finally {
    await client.close();
  }
}

insertData();
```

Output:

```
┌──(hari㉿kali)-[~/Desktop/Advanced-Web-Programming--3161611-/NodeJs/practical-12]
└─$ node "/home/hari/Desktop/Advanced-Web-Programming--3161611-/NodeJs/practical-12/app.js"
6 documents inserted.
```



Practical Quiz:

1. Differentiate between Relational Database and Document Database. **OR** Differentiate between SQL Database and No SQL Database.

Relational Database (SQL):

Structure: Organized into tables with rows and columns.

Schema: Uses predefined schema to define the structure of data.

Data Model: Follows the relational model, with relationships established using foreign keys.

Query Language: Utilizes SQL (Structured Query Language) for querying and manipulating data.

Transactions: Supports ACID (Atomicity, Consistency, Isolation, Durability) transactions.

Scalability: Vertical scaling is common, scaling by increasing the power of hardware.

Examples: MySQL, PostgreSQL, Oracle, SQL Server.

Document Database (NoSQL):

Structure: Stores data in flexible, JSON-like documents.

Schema: Typically schema-less or schema-flexible, allowing each document to have its own structure.

Data Model: Based on key-value pairs or JSON-like documents, allowing nested

structures.

Query Language: Utilizes various query languages or APIs like MongoDB Query Language (MQL).

Transactions: Some NoSQL databases offer eventual consistency instead of strong ACID transactions.

Scalability: Horizontal scaling is typical, scaling out by adding more servers.

Examples: MongoDB, Couchbase, CouchDB.

2. Discuss various methods available in Node JS to perform various operations on MongoDB.

Using MongoDB Native Driver:

Connecting to MongoDB: MongoClient.connect()

Inserting Documents: collection.insertOne(), collection.insertMany()

Querying Documents: collection.find(), collection.findOne()

Updating Documents: collection.updateOne(), collection.updateMany()

Deleting Documents: collection.deleteOne(), collection.deleteMany()

Using Mongoose (ODM for MongoDB):

Defining Schema: mongoose.Schema()

Creating Models: mongoose.model()

Inserting Documents: Model.create(), Model.insertMany()

Querying Documents: Model.find(), Model.findOne()

Updating Documents: Model.updateOne(), Model.updateMany()

Deleting Documents: Model.deleteOne(), Model.deleteMany()

Using MongoDB Atlas (Cloud Database Service):

Creating a Cluster: Set up a MongoDB cluster on MongoDB Atlas.

Connecting to Cluster: Use MongoDB URI to connect from Node.js application.

Performing CRUD Operations: Use MongoDB native driver or Mongoose for CRUD operations.

Using Promises or Async/Await:

Promises: Chain MongoDB operations using .then() and .catch() for error handling.

Async/Await: Use async and await keywords to write asynchronous MongoDB operations more synchronously.

Assessment :

Understanding of Problem (3 marks)	Implementation of Problem (4 marks)	Presentation and report writing (3 marks)	Total (10 marks)

Experiment No: 13

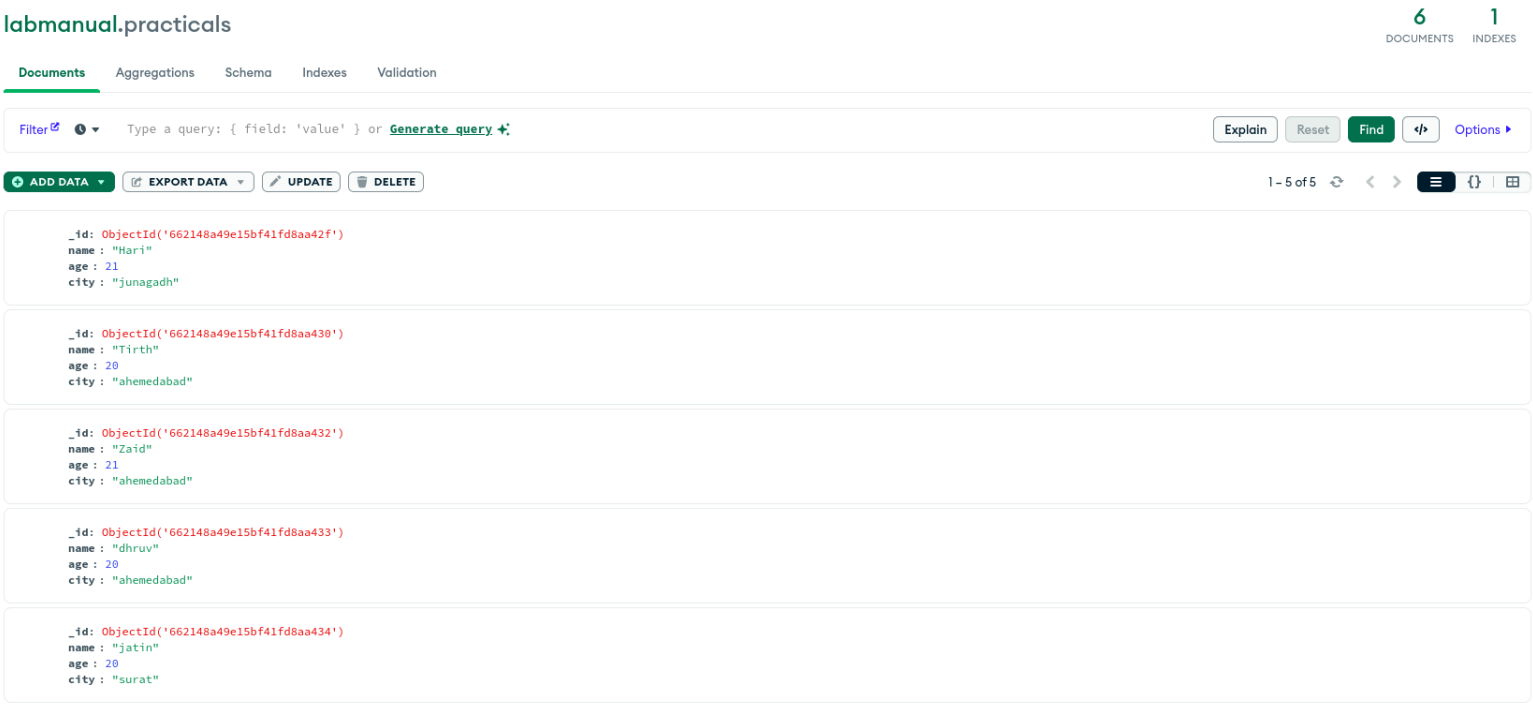
Aim : Write Node JS code to perform deletion operation from Mongo DB.

App.js

```
const { MongoClient } = require('mongodb');

const uri = 'mongodb://127.0.0.1:27017';
const dbName = 'labmanual';
const collectionName = 'practicals';

async function deleteData() {
  const client = new MongoClient(uri);
  try {
    await client.connect();
    const db = client.db(dbName);
    const collection = db.collection(collectionName);

    const query = { name: "Deep" };

    const result = await collection.deleteMany(query);
    console.log(`${result.deletedCount} documents deleted`);
  } catch (error) {
    console.error('Error deleting data:', error);
  } finally {
    await client.close();
  }
}

deleteData();
```

Output:

```
┌──(hari㉿kali)-[~/Desktop/Advanced-Web-Programming--3161611-/NodeJs/practical-13]
└─$ node "/home/hari/Desktop/Advanced-Web-Programming--3161611-/NodeJs/practical-13/app.js"
1 documents deleted
```



Assessment :

Understanding of Problem (3 marks)	Implementation of Problem (4 marks)	Presentation and report writing (3 marks)	Total (10 marks)

Experiment No: 14

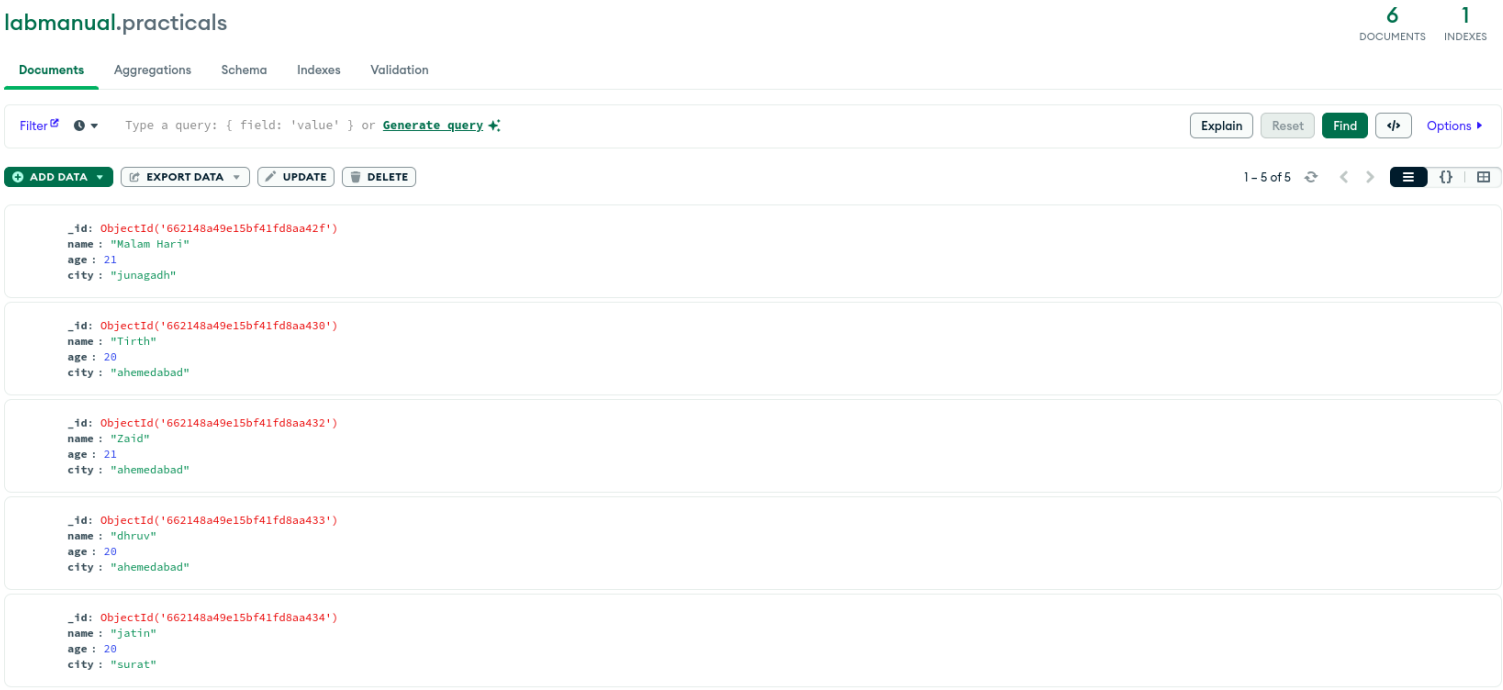
Aim : Write Node JS code to perform selection and updation operation to select and update specific document in Mongo DB.

App.js

```
const { MongoClient } = require("mongodb");
const uri = "mongodb://127.0.0.1:27017";

const dbName = "labmanual";

const collectionName = "practicals";

async function selectAndUpdateData() {
  const client = new MongoClient(uri);
  try {
    await client.connect();
    const db = client.db(dbName);
    const collection = db.collection(collectionName);

    const query = { name: "Hari" };
    const document = await collection.findOne(query);
    console.log("Selected document:", document);

    const filter = { name:"Hari" };
    const update = { $set: { name: "Malam Hari" } };
    const result = await collection.updateOne(filter, update);
    console.log(`${result.modifiedCount} document updated.`);
  } catch (error) {
    console.error("Error:", error);
  } finally {
    await client.close();
  }
}

selectAndUpdateData();
```

Output:

```
┌──(hari㉿kali)-[~/Desktop/Advanced-Web-Programming--3161611-/NodeJs/practical-14]
└─$ node "/home/hari/Desktop/Advanced-Web-Programming--3161611-/NodeJs/practical-14/app.js"
Selected document: {
  _id: new ObjectId('662148a49e15bf41fd8aa42f'),
  name: 'Hari',
  age: 21,
  city: 'junagadh'
}
1 document updated.
```

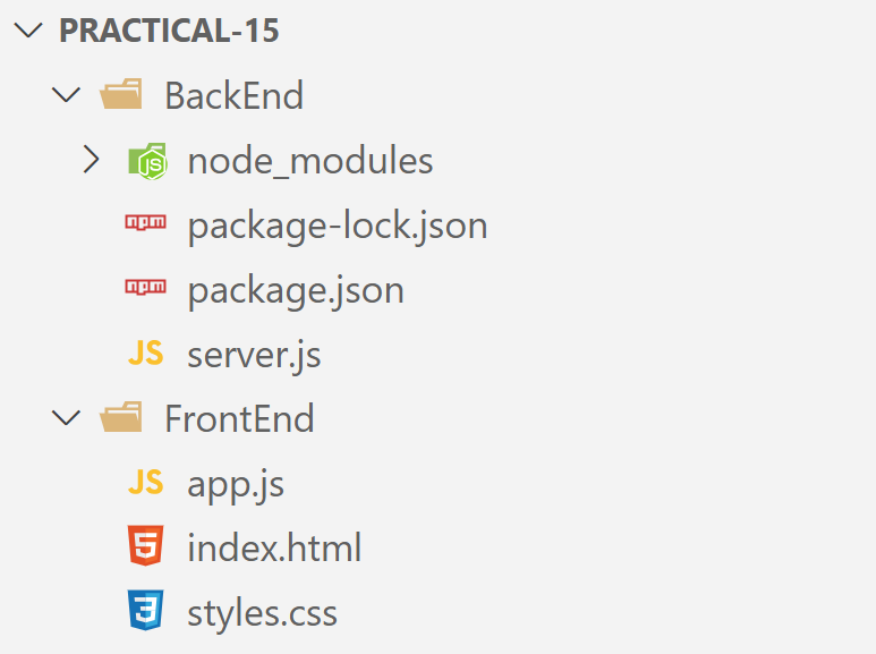


Assessment :

Understanding of Problem (3 marks)	Implementation of Problem (4 marks)	Presentation and report writing (3 marks)	Total (10 marks)

Experiment No: 15

Aim : Create a single page application for Library that will allow the librarian to add a new book and search whether book is currently available in the library or not.



BackEnd/Server.js

```
const express = require("express");
const bodyParser = require("body-parser");
const mongodb = require("mongodb");
const cors = require("cors");

const app = express();
app.use(bodyParser.json());

const mongoURI = "mongodb://127.0.0.1:27017";
const dbName = "library";
app.use(cors());

mongodb.MongoClient.connect(mongoURI)
  .then((client) => {
    console.log("MongoDB connected");
    const db = client.db(dbName);
    const collection = db.collection("books");

    app.post("/api/books", (req, res) => {
      const { title, author } = req.body;
      collection
        .insertOne({ title, author, available: true })
        .then((result) => {
          res.json(result.insertedId);
        })
        .catch((err) => {
          res.status(500).send("Error adding book");
        });
```

```
  });

  app.get("/api/books", (req, res) => {
    collection
      .find({})
      .toArray()
      .then((books) => res.json(books))
      .catch((err) => res.status(500).send("Error fetching books"));
  });

  app.delete("/api/books/:id", (req, res) => {
    const { id } = req.params;
    collection
      .deleteOne({ _id: new mongodb.ObjectId(id) })
      .then((result) => res.json(result))
      .catch((err) => res.status(500).send("Error removing book"));
  });

  const PORT = process.env.PORT || 5000;
  app.listen(PORT, () => console.log(`Server started on port ${PORT}`));
})
.catch((err) => console.error("Error connecting to MongoDB:", err));
```

Index.html

```
<!DOCTYPE html>
<html lang="en" ng-app="libraryApp">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Library App</title>
    <link rel="stylesheet" href="styles.css" />
  </head>
  <body ng-controller="LibraryController">
    <nav>
      <h1>📚 Library Management System</h1>
    </nav>

    <div class="bar">
      <form ng-submit="addBook()">
        <input
          type="text"
          id="title"
          ng-model="newBook.title"
          placeholder="Title"
          required
        />

        <input
          type="text"
          id="author"
          ng-model="newBook.author"
          placeholder="Author"
          required
        />

        <button type="submit" class="add-btn">Add Book</button>
      </form>

      <form ng-submit="searchBook()">
        <input
          type="text"
          ng-model="searchQuery"
          class="search"
          placeholder="Search"
        />
      </form>
```

```
    </div>
    <div class="table">
      <table border="1">
        <thead>
          <tr>
            <th>No.</th>
            <th>Title</th>
            <th>Author</th>
            <th> </th>
          </tr>
        </thead>
        <tbody>
          <tr ng-repeat="book in filteredBooks = (books | filter:searchQuery)">
            <td>{{$index+1}}</td>
            <td>{{ book.title }}</td>
            <td>{{ book.author }}</td>
            <td class="remove">
              <button ng-click="removeBook(book)" class="btn-remove">
                Remove
              </button>
            </td>
          </tr>
        </tbody>
      </table>
    </div>
    <script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.8.2/angular.min.js"></script>
    <script src="app.js"></script>
  </body>
</html>
```

Styles.css

```
*{
  padding: 0;
  margin: 0;
  font-family: 'Courier New', Courier, monospace;
}

nav{
  padding: 20px;
  padding-bottom: 40px;
}

h1{
  text-align: center;
  font-size: 50px;
  color: black;
}

.bar{
  padding: 20px 50px;
  display: flex;
  justify-content: space-between;
  font-size: 30px;
  text-align: center;
}

input[type="text"]{
  font-size: 20px;
  padding: 5px;
  border-radius: 10px;
}

.add-btn{
  font-size: 20px;
  padding: 10px;
  border-radius: 10px;
  background-color: blue;
  color: white;
```

```
  border-color: white;
}

.table{
  padding: 20px 50px;
  display: flex;
  justify-content: center;
}

table{
  border-collapse: collapse;
}


td,th{
  padding: 20px;
  width: 100vw;
  text-align: left;
  font-size: 20px;
}

th{
  font-weight: 700;
  font-size: 25px;
  background-color: rgb(189, 189, 189);
}

tr:nth-child(2n){
  background-color: rgb(235, 235, 235);
}

.btn-remove{
  font-size: 20px;
  font-weight: 600;
  border-color: white;
  padding: 10px;
  border-radius: 10px;
  background-color: red;
  color: white;
}

.remove{
  text-align: center;
}
```

App.js

```
angular.module('libraryApp', [])
    .controller('LibraryController', function($scope, $http) {
        $scope.newBook = {};
        $scope.searchQuery = '';
        $scope.books = [];

        // Function to add a new book
        $scope.addBook = function() {
            $http.post('http://localhost:5000/api/books', $scope.newBook)
                .then(function(response) {
                    console.log('Book added successfully:', response.data);
                    $scope.newBook = {};
                    $scope.fetchBooks();
                })
                .catch(function(error) {
                    console.error('Error adding book:', error);
                });
        };

        $scope.fetchBooks = function() {
```

```
        $http.get('http://localhost:5000/api/books')
            .then(function(response) {
                console.log('Books fetched successfully:', response.data);
                $scope.books = response.data;
            })
            .catch(function(error) {
                console.error('Error fetching books:', error);
            });
    };

    $scope.removeBook = function(book) {
        const bookId = book._id;
        $http.delete(`http://localhost:5000/api/books/${bookId}`)
            .then(function(response) {
                console.log('Book removed successfully:', response.data);
                $scope.fetchBooks();
            })
            .catch(function(error) {
                console.error('Error removing book:', error);
            });
    };
    $scope.fetchBooks();
});
```

Output:

Library Management System

Title | Author | Add Book | Search

No.	Title	Author	
1	Cyber Security	Malam Hari	Remove
2	Adavance Web	Vimal Rathod	Remove
3	AI	vhora saffan	Remove
4	CNS	Prasant Modi	Remove
5	SE	N.V. Nagekar	Remove
6	DE	Rahul Vaja	Remove
7	IPDC	Amit Paramar	Remove

Assessment :

Understanding of Problem (3 marks)	Implementation of Problem (4 marks)	Presentation and report writing (3 marks)	Total (10 marks)