EASWARI ENGINEERING COLLEGE

(Autonomous)

Bharathi Salai, Ramapuram, Chennai-600 089

Department: ______________________________

Name of the Lab (with code) ______________________

Name	:
Reg No.	:
Semester	:
Year	:
Branch	:

EASWARI ENGINEERING COLLEGE

(Autonomous)

Bharathi Salai, Ramapuram, Chennai-600 089

Department: ________________________________

PRACTICAL EXAMINATIONS_______________ (Month/Year)

BONAFIDE CERTIFICATE

This is to Certify that this practical work titled___________________

(code)

____is the bonafide work of

(Name of the Laboratory)

Mr./Miss.____________________________ (Name of the Student)

With Registration Number________________________________

______ in semester_________________of year in the Department of

_______________________________________during the academic year 20____

20___.

Faculty Incharge Head of the Department

Submitted for Practical Examination held on_____/_____/__________at

Easwari Engineering College,Ramapuram,Chennai-89

Internal Examiner External Examiner

Sl No	DATE	CONTENT	PAGE NO	SIGNATURE
1		Create a form and validate the contents of the form using JavaScript.		
2		Get data using Fetch API from an open-source endpoint and display the contents in the form of a card		
3		Create a NodeJS server that serves static HTML and CSS files to the user without using Express.		
4		Create a NodeJS server using Express that stores data from a form as a JSON file and dis-plays it in another page. The redirect page should be prepared using Handlebars.		
5		Create a NodeJS server using Express that creates, reads, updates and deletes students' details and stores them in MongoDB database. The information about the user should be obtained from a HTML form.		
6		Create a NodeJS server that creates, reads, updates and deletes event details and stores them in a MySQL database. The information about the user should be obtained from a HTML form.		
7		Create a counter using ReactJS		
8		Create a Todo application using ReactJS. Store the data to a JSON file using a simple NodeJS server and retrieve the information from the same during page reloads.		
9		Create a simple Sign up and Login mechanism and authenticate the user using cookies. The user information can be stored in either MongoDB or MySQL and the server should be built using NodeJS and Express Framework		
10		Create and deploy a virtual machine using a virtual box that can be accessed from the host computer using SSH.		
11		Create a docker container that will deploy a NodeJS ping server using the NodeJS image.		

Ex.No:1a	**Create a form and validate the contents of the form using javascript**
Date:	

Aim:

To create a **user registration form** with input fields for name, email, password, and age and validate the user input using **JavaScript** to ensure the correctness of data before submission.

Algorithm:

Step 1: Create the HTML Structure

- Define a <form> with input fields for:
  - Name
  - Email
  - Password
  - Age
  - A **Submit** button
- Add <div> elements for **error messages** below each input field.

Step 2: Apply CSS for Styling

- Set a **flexbox layout** to center the form.
- Style the input fields with proper **padding**, **borders**, and **spacing**.
- Add **error messages** in red to indicate invalid inputs.
- Style the submit button.

Step 3: Implement JavaScript Validation

1. **Prevent Default Form Submission:**
   - Add an eventListener to the **form's submit** event.
   - Use event.preventDefault() to stop submission if validation fails.
2. **Validate Each Input Field:**
   - **Name:**
     - Ensure the length is **at least 3 characters**.
   - **Email:**
     - Check if it follows a **valid email format** using a **regular expression (regex)**
   - **Password:**
     - Ensure the password has **at least 6 characters**.
   - **Age:**
     - Ensure the user is **at least 18 years old**.
3. **Display Error Messages:**
   - If a field is invalid, show the **error message** next to it.
   - Clear **previous errors** before running validation again.

4. **Submit the Form If Valid:**
   - If all inputs pass validation, **show an alert** message.
   - Reset the form fields using form.reset().

Program

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
<meta name="viewport" content="width=device-width, initial-scale=1.0">
<title>Form Validation</title>
<style>
        body {
            font-family: Arial, sans-serif;
            background-color: #f4f4f4;
            display: flex;
            justify-content: center;
            align-items: center;
            height: 100vh;
            margin: 0;
        }
        .container {
           background: white;
           padding: 20px;
           border-radius: 8px;
            box-shadow: 0 4px 8px rgba(0, 0, 0, 0.1);
            width: 300px;
            text-align: center;
        }
        input {
            width: 100%;
            padding: 10px;
            margin: 10px 0;
            border: 1px solid #ccc;
            border-radius: 5px;
        }
.error {
color: red;
            font-size: 14px;
            text-align: left;
        }
        button {
            background-color: blue;
color: white;
            border: none;
            padding: 10px;
            cursor: pointer;
            width: 100%;
            border-radius: 5px;
        }
button:hover {
            background-color: darkblue;
        }
</style>
</head>
<body>

<div class="container">
```

```
<h2>Registration Form</h2>
<form id="myForm">
<input type="text" id="name" placeholder="Full Name">
<div class="error" id="nameError"></div>

<input type="email" id="email" placeholder="Email">
<div class="error" id="emailError"></div>

<input type="password" id="password" placeholder="Password">
<div class="error" id="passwordError"></div>

<input type="number" id="age" placeholder="Age">
<div class="error" id="ageError"></div>

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

<script>
document.getElementById("myForm").addEventListener("submit", function(event) {
event.preventDefault(); // Prevent form submission

        let isValid = true;

        // Clear previous errors
document.getElementById("nameError").innerText = "";
document.getElementById("emailError").innerText = "";
document.getElementById("passwordError").innerText = "";
document.getElementById("ageError").innerText = "";
```

```
        // Get input values
        let name = document.getElementById("name").value.trim();
        let email = document.getElementById("email").value.trim();
        let password = document.getElementById("password").value.trim();
        let age = document.getElementById("age").value.trim();

        // Name validation (at least 3 characters)
        if (name.length< 3) {
document.getElementById("nameError").innerText = "Name must be at least 3 characters.";
isValid = false;
        }

        // Email validation (simple regex)
        let emailPattern = /^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$/;
        if (!emailPattern.test(email)) {
document.getElementById("emailError").innerText = "Enter a valid email.";
isValid = false;
        }

        // Password validation (at least 6 characters)
        if (password.length< 6) {
document.getElementById("passwordError").innerText = "Password must be at least 6 characters.";
isValid = false;
        }

        // Age validation (must be at least 18)
        if (age === "" || isNaN(age) || age < 18) {
document.getElementById("ageError").innerText = "You must be at least 18 years old.";
isValid = false;
            }
            // If all fields are valid, submit the form
            if (isValid) {
alert("Form submitted successfully!");
document.getElementById("myForm").reset();
            }
        });
</script>

</body>
</html>
```

Software Required:

1. **Text Editor / IDE (for writing code)**
   - **VS Code**
   - Sublime Text
   - Notepad++
   - Atom
   - Any basic text editor (Notepad, nano, etc.)
2. **Web Browser (for running the code)**
   - **Google Chrome**
   - Mozilla Firefox
   - Microsoft Edge
   - Safari
   - Any modern web browser

Result:

Thus the program has been executed successfully and output verified.

Ex.No:1b	**Create a form and validate the contents of the form using javascript**
Date:	

Aim:

To collect user information (Name, Email, Department, Age) via a form and validate the inputs before submission.

Algorithm:

- Display a form with Name, Email, Department, and Age fields.
- On form submission, trigger validateForm() function.
- Check if all fields are filled.
- Validate the email format.
- If valid, allow submission. If not, show alerts and block submission.

Program:

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
<meta name="viewport" content="width=device-width, initial-scale=1.0">
<title>Simple Form</title>
</head>
<body>

<h2>User Information Form</h2>
<form id="userForm" onsubmit="return validateForm()">
<label for="name">Name:</label><br>
<input type="text" id="name" name="name"><br>
<label for="email">Email:</label><br>
<input type="email" id="email" name="email"><br>
```

```
<label for="department">Department:</label><br>
<input type="text" id="department" name="department"><br>

<label for="age">Age:</label><br>
<input type="number" id="age" name="age"><br>
<input type="submit" value="Submit">
</form>

<script>
   function validateForm() {
const name = document.getElementById("name").value;
const email = document.getElementById("email").value;
const department=document.getElementById("department").value;
const age=document.getElementById("age").value;

       if (!name) {
alert("Name is required.");
           return false;
       }

       if (!email) {
alert("Email is required.");
           return false;
       }
       if (!department) {
alert("Department is required.");
         return false;
       }
       if (!age) {
```

```
alert("Age is required.");
            return false;
        }
               return true; // Allow form submission
    }
</script>
</body>
</html>
```

Explanation:

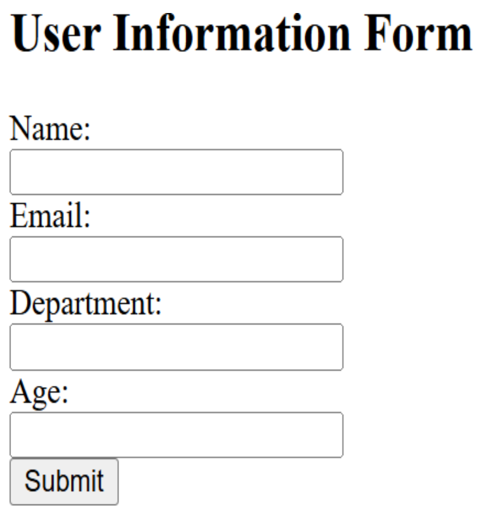
Form Fields:

- Name, Email, Department, and Age are input fields where users can type their information.
- The Age field uses a number input type, which restricts the user to enter only numeric values.

Validation:

- The validateForm() JavaScript function checks if all fields are filled.
- It also uses a regular expression to check if the email follows a valid format.
- If all inputs are valid, the form is submitted. Otherwise, an alert is shown indicating what is missing.

Output:


User Information Form

Name:

Email:

Department:

Age:

Result:

Thus the program has been executed successfully and output verified.

Ex.No:2	**Fetch data using the Fetch API from an open-source endpoint and display the contents in the form of a chart.**
Date:	

Aim:

To retrieve data from an open-source API using JavaScript's Fetch API and visualize the data in a graphical format using Chart.js.

Steps:

- Choose an open-source API that provides numerical data suitable for chart representation.
- Use the Fetch API to retrieve data asynchronously.
- Process and extract relevant data from the API response.
- Use Chart.js to create a chart based on the retrieved data.
- Display the chart dynamically on a webpage.

Program :

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Fetch API Example</title>
    <style>
        body {
            font-family: Arial, sans-serif;
            display: flex;
            justify-content: center;
            align-items: center;
            height: 100vh;
            background-color: #f4f4f4;
        }
        .card {
            width: 300px;
            padding: 20px;
            background: white;
            border-radius: 10px;
            box-shadow: 0px 4px 6px rgba(0, 0, 0, 0.1);
            text-align: center;
        }
        img {
            width: 100px;
            height: 100px;
            border-radius: 50%;
            margin-bottom: 10px;
        }
        button {
            padding: 10px;
            background: #007BFF;
            color: white;
            border: none;
            border-radius: 5px;
            cursor: pointer;
        }
        button:hover {
            background: #0056b3;
        }
    </style>
</head>
<body>

    <div class="card">
    <img id="userImage" src="" alt="User Image">
    <h3 id="userName">Loading...</h3>
    <p id="userEmail"></p>
    <button onclick="fetchUser()">Get New User</button>
</div>

<script>
    async function fetchUser() {
        try {
            console.log("Fetching user data...");
            const response = await fetch('https://randomuser.me/api/');

            if (!response.ok) {
                throw new Error(`HTTP error! Status: ${response.status}`);
            }

            const data = await response.json();
            console.log("Data received:", data); // Debugging log

            const user = data.results[0];

            document.getElementById('userImage').src = user.picture.large;
            document.getElementById('userName').textContent = `${user.name.first} ${user.name.last}`;
            document.getElementById('userEmail').textContent = user.email;
        } catch (error) {
            console.error('Error fetching data:', error);
            document.getElementById('userName').textContent = "Failed to load user.";
        }
    }
```

```
        // Fetch a user when the page loads
        window.onload = fetchUser;
    </script>

</body>
</html>
```

Result:

Thus the program has been executed successfully and output verified.

Ex.No:3	Create a NodeJS server that serves static HTML and CSS files to the user without using express
Date:	

Aim:

To develop a Node.js server that serves static HTML and CSS files to the user without using any external frameworks like Express.js.

Algorithm:

1. Initialize the project:
   - o Create a new directory for the project and navigate into it.
   - o Add the necessary files: index.html, style.css, and server.js.
2. Write the HTML and CSS:
   - o Create an HTML file (index.html) containing the structure of the webpage.
   - o Create a CSS file (style.css) for styling the HTML content.
3. Create the server in Node.js:
   - o Import required modules (http, fs, and path).
   - o Create an HTTP server using the http.createServer() method.
   - o Handle incoming requests:
     - ♣ Determine the requested file (index.html for /, or the file based on the URL).
     - ♣ Identify the file extension and set the appropriate MIME type.
   - o Read and serve the requested file:
     - ♣ Use fs.readFile() to read the file's content.
     - ♣ Send the content back with the correct Content-Type.
     - ♣ Handle errors, such as file not found (404).
4. Run the server:
   - o Start the server on a specified port (e.g., 3000).
   - o Test the server by accessing it in a browser.

Program:

index.html:

```
<!DOCTYPE html>
<html lang="en">
	<head>
		<meta charset="UTF-8">
		<meta name="viewport" content="width=device-width, initial-scale=1.0">
		<title>Static Server</title>
		<link rel="stylesheet" href="/style.css">
	</head>
	<body>
		<h1>Welcome to My Static Server</h1>
		<p>This is a simple server serving static HTML and CSS files.</p>
	</body>
</html>
```

style.css:

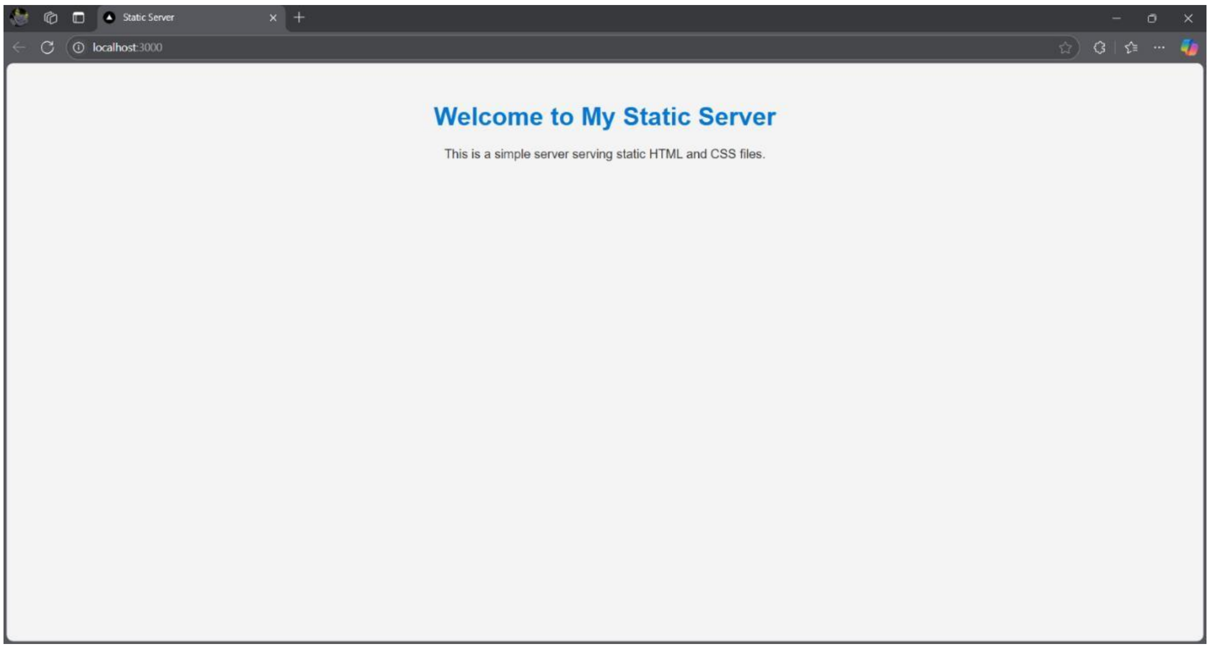
```
body {
	font-family: Arial, sans-serif;
	text-align: center;
	margin: 50px;
	background-color: #f4f4f4;
	color: #333;
}
h1 {
	color: #0078D7;
}
```

server.js:

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

const server = http.createServer((req, res) => {
        let filePath = '.' + req.url;
         if (filePath === './') {
                filePath = './index.html';
        }

        constextname = path.extname(filePath);
        constmimeTypes = {
                '.html': 'text/html',
                '.css': 'text/css',
         };

        constcontentType = mimeTypes[extname] || 'application/octet-stream';

        fs.readFile(filePath, (err, content) => {
        if (err) {
        if (err.code === 'ENOENT') {
        res.writeHead(404, { 'Content-Type': 'text/html' });
        res.end('<h1>404 Not Found</h1>', 'utf-8');
        } else {
                res.writeHead(500);
                res.end(`Server Error: ${err.code}`, 'utf-8');
        }
```

```
            } else {
                res.writeHead(200, { 'Content-Type': contentType });
                res.end(content, 'utf-8');
            }
  });
});


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

Sample output:



Welcome to My Static Server

This is a simple server serving static HTML and CSS files.

Requirements:

1. **Hardware**:
   - A computer with Node.js installed.
2. **Software**:
   - Node.js (runtime environment).
   - A text editor (e.g., Visual Studio Code, Sublime Text).
   - Web browser (e.g., Chrome, Firefox).
3. **Files**:
   - index.html: HTML file for the webpage.
   - style.css: CSS file for styling.
   - server.js: Node.js script to create and run the server.

Result:

A Node.js server successfully serves static HTML and CSS files.

Ex.No:4	Create a Node.js server using Express that stores data from a form as a JSON file and displays it on another page. The redirected page should be prepared using Handlebars.
Date:	

Aim:

To develop a **Node.js server using Express** that collects form data, stores it in a JSON file, and displays the stored data on another page using Handlebars as the templating engine.

Algorithm:

1. **Initialize Project**: Create a Node.js project and install required dependencies (express, express-handlebars, body-parser, fs).
2. **Setup Express Server**: Initialize Express and configure middleware (body-parser for form data parsing).
3. **Configure Handlebars**: Set up Handlebars as the view engine.
4. **Create Routes**:
    a. **GET /form**: Render an HTML form using Handlebars.
    b. **POST /submit**:
        i. Receive form data.
        ii. Store it in a JSON file (data.json).
        iii. Redirect to the display page.
    c. **GET /display**:
        i. Read data from data.json.
        ii. Render it using Handlebars.
5. **Start Server**: Listen on a specified port.

Code:

Views/layouts/main.handlebars

```
<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, initial-scale=1.0">
```

```
<title>{{title}}</title>
</head>
<body>
  {{{body}}} <!-- This is where the content of the views will be injected -->
</body>
</html>
```

Views/display.handlebars

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
<meta name="viewport" content="width=device-width, initial-scale=1.0">
<title>Display Data</title>
</head>
<body>
<h1>Submitted Data</h1>
<p>Name: {{data.name}}</p>
<p>Email: {{data.email}}</p>
<a href="/">Go Back</a>
</body>
</html>
```

views/form.handlebars

```
<!-- FSWD/exp4/views/form.handlebars -->
<!DOCTYPE html>
<html lang="en">
```

```
<head>
<meta charset="UTF-8">
<meta name="viewport" content="width=device-width, initial-scale=1.0">
<title>Form</title>
</head>

<body>
<h1>Submit Your Data</h1>
<form id="dataForm">
<label for="name">Name:</label>
<input type="text" id="name" name="name" required>
<br>
<label for="email">Email:</label>
<input type="email" id="email" name="email" required>
<br>
<button type="submit">Submit</button>
</form>

<script>
document.getElementById('dataForm').addEventListener('submit', function (event) {
event.preventDefault(); // Prevent the default form submission

constformData = {
        name: document.getElementById('name').value,
        email: document.getElementById('email').value
      };

fetch('/submit', {
        method: 'POST',
                headers: {
                    'Content-Type': 'application/json'
                },
                body: JSON.stringify(formData)
            })
.then(response => {
                    if (response.redirected) {
window.location.href = response.url; // Redirect to the display page
                    }
                });
        });
</script>
</body>

</html>
```

data.json

```
{
  "name": "kishana",
  "email": "hayagriva.21@stu.srmuniversity.ac.in"
}
```

Index.js

```
const express = require("express");
const fs = require("fs");
const path = require("path");
constexphbs = require("express-handlebars").engine;
```

```
const app = express();
const PORT = 3000;

// Set up Handlebars as the view engine
app.engine("handlebars", exphbs());
app.set("view engine", "handlebars");

// Middleware to parse JSON data
app.use(express.json());
app.use(express.static("public"));

// Route to display the form
app.get("/", (req, res) => {
res.render("form");
});

// Route to handle form submission
app.post("/submit", (req, res) => {
constformData = req.body;

  // Save form data to a JSON file
fs.writeFileSync(
path.join(__dirname, "data.json"),
JSON.stringify(formData, null, 2)
  );

  // Redirect to the display page
res.redirect("/display");
});
```

```
// Route to display the stored data
app.get("/display", (req, res) => {
const data = JSON.parse(
fs.readFileSync(path.join(__dirname, "data.json"), "utf-8")
  );
res.render("display", { data });
});

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

OUTPUT:

Result:

The program is executed successfully.

Ex.No:5	**Create a NodeJS server using Express that creates, reads, updates and deletes students' details and stores them in MongoDB database. The information about the user should be obtained from a HTML form.**
Date:	

Aim:

To develop a **Student Management System** using **Node.js, Express.js, and MongoDB** that allows users to **add, update, delete, and view student records** through an HTML form.

Procedure:

1. **Install Dependencies**
   a. Shell or terminal :

      npminit -y

      npm install express mongoose body-parser corsejs

2. **Set Up Express Server**

   Connect to MongoDB

   Define Student Schema

   Create API routes for CRUD operations

3. **Create Views** (EJS Templates)

   Form to add students

   Table to display student records

4. **Run the Server**

   Shell or terminal :node server.js

5. **Access the Application**

   Open **http://localhost:5000** in a browser

Code:

Views/index.ejs:

```
<!DOCTYPE html>

<html lang

="en">
```

```
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Student Management</title>
    <style>
        body { font-family: Arial, sans-serif; text-align: center; }
        table { width: 50%; margin: 20px auto; border-collapse: collapse; }
        table, th, td { border: 1px solid black; padding: 10px; }
        form { margin: 20px auto; width: 50%; }
        input, button { padding: 8px; margin: 5px; }
    </style>
</head>
<body>
    <h2>Student Management</h2>
<!-- Add Student Form -->
    <form action="/add" method="POST">
        <input type="text" name="name" placeholder="Name" required>
        <input type="number" name="age" placeholder="Age" required>
        <input type="email" name="email" placeholder="Email" required>
        <button type="submit">Add Student</button>
    </form>

    <!-- Student List -->
    <h3>Student Records</h3>
    <table>
        <tr>
            <th>Name</th>
            <th>Age</th>
            <th>Email</th>
            <th>Actions</th>
        </tr>
        <% students.forEach(student => { %>
        <tr>
            <td><%= student.name %></td>
            <td><%= student.age %></td>
            <td><%= student.email %></td>
            <td>
                <form action="/update/<%= student._id %>" method="POST" style="display:inline;">
                    <input type="text" name="name" value="<%= student.name %>" required>
                    <input type="number" name="age" value="<%= student.age %>" required>
                    <input type="email" name="email" value="<%= student.email %>" required>
                    <button type="submit">Update</button>
                </form>
                <a href="/delete/<%= student._id %>" style="color:red;">Delete</a>
            </td>
        </tr>
        <% }) %>
    </table>
</body>
</html>
```

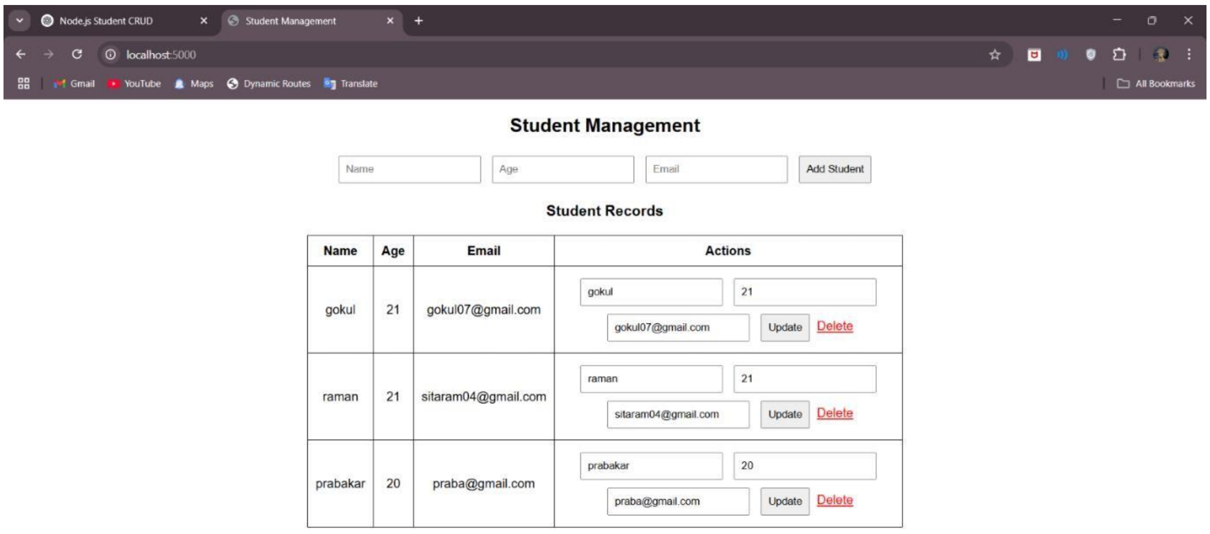
server.js:

```
const express = require("express");
const mongoose = require("mongoose");
constbodyParser = require("body-parser");
constcors = require("cors");
const app = express();
const PORT = 5000;
```

```
// Middleware
app.use(cors());
app.use(bodyParser.urlencoded({ extended: true }));
app.use(bodyParser.json());
app.set("view engine", "ejs");

// Connect to MongoDB
mongoose.connect("mongodb://127.0.0.1:27017/studentsDB", {
    useNewUrlParser: true,
    useUnifiedTopology: true,
}).then(() => console.log("Connected to MongoDB"))
  .catch(err => console.log(err));

// Student Schema
conststudentSchema = new mongoose.Schema({
    name: String,
    age: Number,
    email: String
});
const Student = mongoose.model("Student", studentSchema);

// Home Page (Displays Students)
app.get("/", async (req, res) => {
    const students = await Student.find();
    res.render("index", { students });
});

// Create Student (Form Submission)
```

```
app.post("/add", async (req, res) => {
    const{ name, age, email } = req.body;
    constnewStudent = new Student({ name, age, email });
    await newStudent.save();
    res.redirect("/");
});

// Update Student (Form Submission)
app.post("/update/:id", async (req, res) => {
    const{ name, age, email } = req.body;
    await Student.findByIdAndUpdate(req.params.id, { name, age, email });
    res.redirect("/");
});

// Delete Student
app.get("/delete/:id", async (req, res) => {
    await Student.findByIdAndDelete(req.params.id);
    res.redirect("/");
});

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

Sample output:



Student Management

Name | Age | Email | Add Student

Student Records

Name	Age	Email	Actions
gokul	21	gokul07@gmail.com	gokul 21 gokul07@gmail.com Update Delete
raman	21	sitaram04@gmail.com	raman 21 sitaram04@gmail.com Update Delete
prabakar	20	praba@gmail.com	prabakar 20 praba@gmail.com Update Delete

Software Requirements:

- Node.js
- MongoDB
- Express.js
- EJS (Embedded JavaScript Templates)
- Mongoose (MongoDB ORM)
- Body-Parser (for form data processing)
- Cors (for cross-origin requests)

Result:

A Node.js server successfully serves static HTML and CSS files.

Ex.No:6	**Create a NodeJS server that creates, reads, updates and deletes event details and stores them in a MySQL database. The information about the user should be obtained from a HTML form.ate a NodeJS server using Express that creates, reads, updates and deletes students' details and stores them in MongoDB database. The information about the user should be obtained from a HTML form.**
Date:	

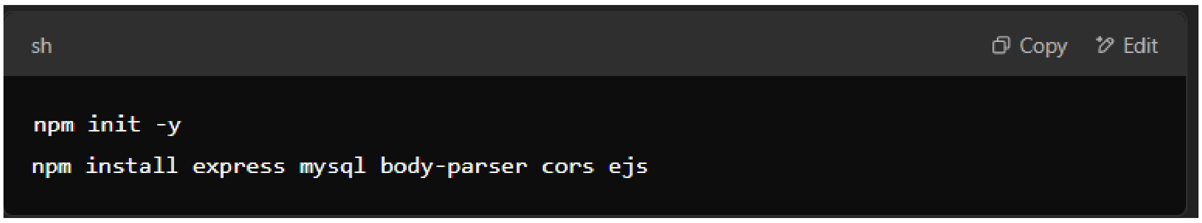
Aim:

To develop an **Event Management System** using **Node.js, Express.js, and MySQL** that allows users to **add, update, delete, and view event records** through an HTML form.

Procedure:

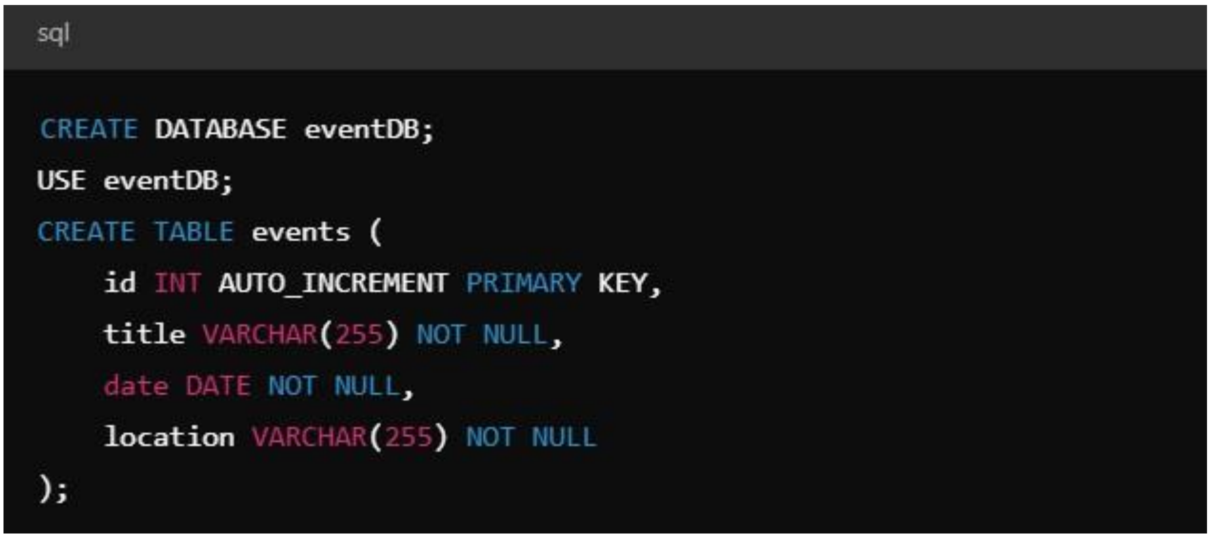
Procedure:

1. **Install Dependencies**



```sh
npm init -y
npm install express mysql body-parser cors ejs
```

2. **Set Up MySQL Database**



```sql
CREATE DATABASE eventDB;
USE eventDB;
CREATE TABLE events (
    id INT AUTO_INCREMENT PRIMARY KEY,
    title VARCHAR(255) NOT NULL,
    date DATE NOT NULL,
    location VARCHAR(255) NOT NULL
);
```

3. **Create Node.js Server**
   a. Connect to MySQL

    b. Define routes for CRUD operations

4. **Create Views (EJS Templates)**
    a. Form to add events
    b. Table to display event records
5. **Run the Server**

```sh
node server.js
```

6. **Access the Application**

    Open **http://localhost:5000** in a browser

Code:

Views/index.ejs:

```
<!DOCTYPE html>
<html lang="en">
<head>
   <meta charset="UTF-8">
   <meta name="viewport" content="width=device-width, initial-scale=1.0">
   <title>Event Management</title>
   <style>
      body { font-family: Arial, sans-serif; text-align: center; }
      table { width: 60%; margin: 20px auto; border-collapse: collapse; }
      table, th, td { border: 1px solid black; padding: 10px; }
      form { margin: 20px auto; width: 50%; }
      input, button { padding: 8px; margin: 5px; }
   </style>
</head>
<body>
   <h2>Event Management</h2>
```

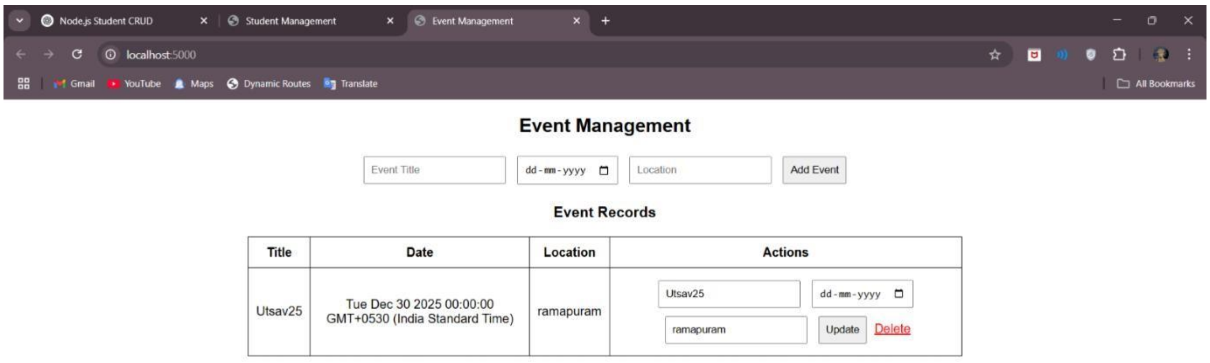
```
<!-- Add Event Form -->
<form action="/add" method="POST">
    <input type="text" name="title" placeholder="Event Title" required>
    <input type="date" name="date" required>
    <input type="text" name="location" placeholder="Location" required>
    <button type="submit">Add Event</button>
</form>

<!-- Event List -->
<h3>Event Records</h3>
<table>
    <tr>
        <th>Title</th>
        <th>Date</th>
        <th>Location</th>
        <th>Actions</th>
    </tr>
    <% events.forEach(event => { %>
    <tr>
        <td><%= event.title %></td>
        <td><%= event.date %></td>
        <td><%= event.location %></td>
        <td>
            <form action="/update/<%= event.id %>" method="POST" style="display:inline;">
                <input type="text" name="title" value="<%= event.title %>" required>
                <input type="date" name="date" value="<%= event.date %>" required>
                <input type="text" name="location" value="<%= event.location %>" required>
                <button type="submit">Update</button>
                </form>
                <a href="/delete/<%= event.id %>" style="color:red;">Delete</a>
            </td>
        </tr>
        <% }) %>
    </table>
</body>
</html>
```

Server.js:

```
const express = require("express");
constmysql = require("mysql");
constbodyParser = require("body-parser");
constcors = require("cors");
const path = require("path");

const app = express();
const PORT = 5000;

// Middleware
app.use(cors());
app.use(bodyParser.urlencoded({ extended: true }));
app.use(bodyParser.json());
app.set("view engine", "ejs");
app.set("views", path.join(__dirname, "views"));

// MySQL Connection
constdb = mysql.createConnection({
    host: "localhost",
    user: "root",  // Change this if you have a different MySQL user
    password: "123",  // Set your MySQL password
    database: "eventDB",
});

db.connect((err) => {
    if (err) throw err;
    console.log("Connected to MySQL Database");
});

// Home Page - Display Events
app.get("/", (req, res) => {
    constsql = "SELECT * FROM events";
    db.query(sql, (err, results) => {
        if (err) throw err;
        res.render("index", { events: results });
    });
});

// Create Event
app.post("/add", (req, res) => {
    const{ title, date, location } = req.body;
    constsql = "INSERT INTO events (title, date, location) VALUES (?, ?, ?)";
    db.query(sql, [title, date, location], (err) => {
        if (err) throw err;
        res.redirect("/");
    });
});
```

```
// Update Event
app.post("/update/:id", (req, res) => {
    const{ title, date, location } = req.body;
    constsql = "UPDATE events SET title=?, date=?, location=? WHERE id=?";
    db.query(sql, [title, date, location, req.params.id], (err) => {
        if (err) throw err;
        res.redirect("/");
    });
});

// Delete Event
app.get("/delete/:id", (req, res) => {
    constsql = "DELETE FROM events WHERE id=?";
    db.query(sql, [req.params.id], (err) => {
        if (err) throw err;
        res.redirect("/");
    });
});

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

Sample output:



Event Management

Event Title | dd-mm-yyyy | Location | Add Event

Event Records

Title	Date	Location	Actions
Utsav25	Tue Dec 30 2025 00:00:00 GMT+0530 (India Standard Time)	ramapuram	Utsav25 dd-mm-yyyy ramapuram Update Delete

Software Requirements:

- Node.js
- MySQL
- Express.js
- EJS (Embedded JavaScript Templates)
- Body-Parser (for form data processing)
- Cors (for cross-origin requests

Result:

A Node.js server successfully serves static HTML and CSS files.

Ex.No:7	Create a counter using ReactJS
Date:	

Aim:

To develop a **ReactJS-based counter application** that allows users to increment, decrement, and reset a counter value dynamically.

Algorithm:

1. **Initialize Project**: Create a React app using create-react-app or Vite.
2. **Setup Component**: Create a Counter component.
3. **Define State**: Use useState to manage the counter value.
4. **Implement Functions**:
   a. **Increment**: Increase counter by 1.
   b. **Decrement**: Decrease counter by 1 (ensure it doesn't go below zero if required).
   c. **Reset**: Set counter back to zero.
5. **Render UI**: Display the counter value with three buttons (**Increment, Decrement, Reset**).
6. **Handle Events**: Attach event handlers to buttons to update the counter state.
7. **Run and Test**: Start the app and test its functionality.

Code:

src/App.jsx

```
import { useState } from 'react'

import reactLogo from './assets/react.svg'

import viteLogo from '/vite.svg'

import './App.css'


function App() {

const [count, setCount] = useState(0)


  return (

<>

<div>

<a href="https://vite.dev" target="_blank">
<imgsrc={viteLogo} className="logo" alt="Vite logo" />
</a>
<a href="https://react.dev" target="_blank">
<imgsrc={reactLogo} className="logo react" alt="React logo" />
</a>
</div>
<h1>Vite + React</h1>
<div className="card">
<button onClick={() =>setCount((count) => count + 1)}>
        count is {count}
</button>
</div>
</>
  )
}

export default App
```

Main/jsx

```
import { StrictMode } from 'react'
import { createRoot } from 'react-dom/client'
import './index.css'
import App from './App.jsx'

createRoot(document.getElementById('root')).render(
<StrictMode>
<App />
</StrictMode>,
)
```

Result:

Thus the program has been executed successfully and output verified.

Ex.No:8	**Create a Todo application using react.js . store the data to a JSON file using a simple NodeJs server and retrieve the information from the same during page reloads**
Date:	

Aim :

To develop a simple Todo application with React as the frontend and Node.js with Express as the backend, enabling users to create, view, and delete todos, with data being persisted in a JSON file.

Procedure :

1. **Setup Backend:**
   - Install express, cors, body-parser, and fs using npm.
   - Create the server.js file for the backend.
   - Implement REST API endpoints (GET, POST, and DELETE) for managing todos.
   - Use a JSON file (todos.json) for data storage.
2. **Setup Frontend:**
   - Create a React app using create-react-app or any setup method.
   - Install axios to handle HTTP requests.
   - Implement React components and hooks (useState, useEffect) for managing state and fetching data from the backend.
   - Create UI elements such as input fields, buttons, and lists for interacting with todos.
3. **Run the Application:**
   - Start the backend server (node server.js).
   - Start the React frontend (npm start).
   - Interact with the Todo app via the frontend, which communicates with the backend.

Program :

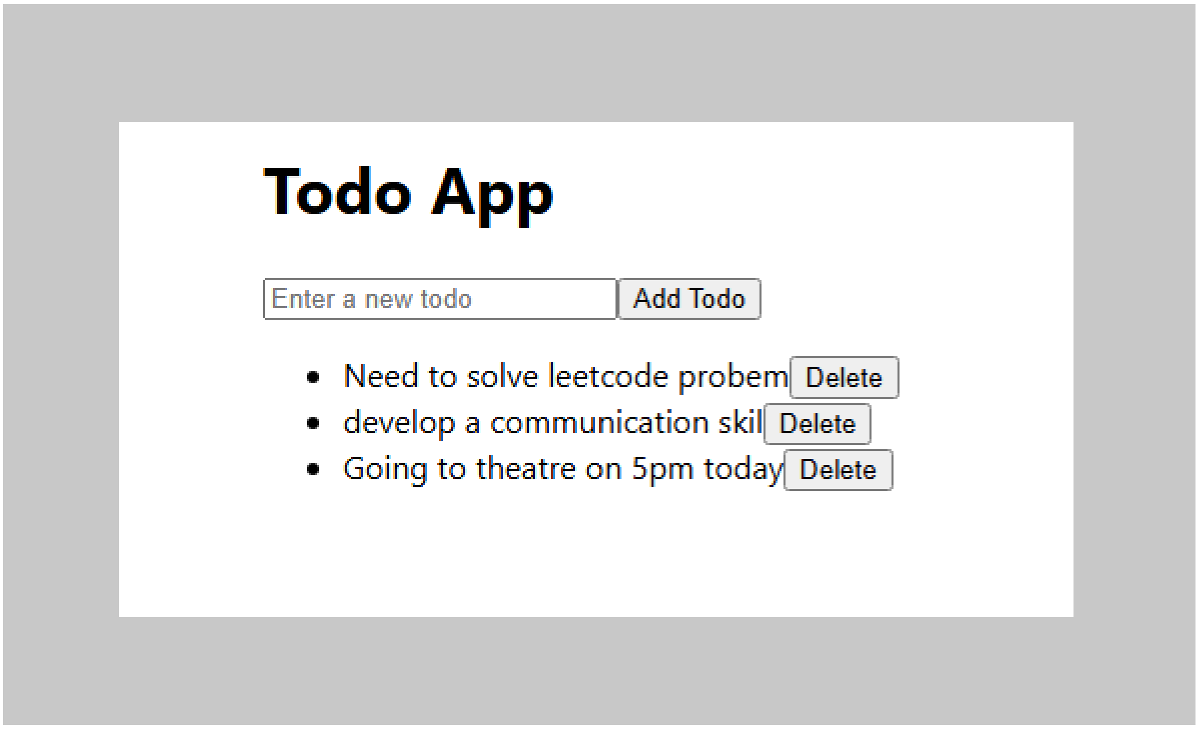
Frontend (React - App.js**)**

```
importReact, { useState, useEffect } from'react';
importaxiosfrom'axios';
import'./App.css';

constApp = () => {
const [todos, setTodos] = useState([]);
const [newTodo, setNewTodo] = useState('');

constfetchTodos = async () => {
try {
const response = awaitaxios.get('http://localhost:5000/todos');
setTodos(response.data);
    } catch (error) {
console.error('Error fetching todos:', error);
    }
  };
useEffect(() => {
fetchTodos();
  }, []);

consthandleAddTodo = async () => {
if (newTodo.trim()) {
consttodo = { id: Date.now().toString(), text: newTodo };
try {
awaitaxios.post('http://localhost:5000/todos', todo);
setTodos([...todos, todo]);
setNewTodo('');
      } catch (error) {
console.error('Error adding todo:', error);
      }
    }
  };

consthandleDeleteTodo = async (id) => {
try {
awaitaxios.delete(`http://localhost:5000/todos/${id}`);
setTodos(todos.filter(todo => todo.id !== id));
    } catch (error) {
console.error('Error deleting todo:', error);
    }
  };

return (
<div className="App">
<h1>Todo App</h1>
<input
        type="text"
        value={newTodo}
onChange={(e) =>setNewTodo(e.target.value)}
        placeholder="Enter a new todo"
      />
<button onClick={handleAddTodo}>Add Todo</button>
<ul>
        {todos.map(todo => (
 <li key={todo.id}>
            {todo.text}
<button onClick={() =>handleDeleteTodo(todo.id)}>Delete</button>
</li>
        ))}
</ul>
</div>
  );
};

exportdefaultApp;
```

Backend (Node.js - server.js**)**

javascript
CopyEdit
```
const express = require('express');
constcors = require('cors');
constbodyParser = require('body-parser');
const fs = require('fs');
const path = require('path');

const app = express();
constPORT = process.env.PORT || 5000;

app.use(cors());
app.use(bodyParser.json());

consttodosFilePath = path.join(__dirname, 'todos.json');

constgetTodos = () => {
if(!fs.existsSync(todosFilePath)) {
fs.writeFileSync(todosFilePath, JSON.stringify([]));
  }
const data = fs.readFileSync(todosFilePath);
returnJSON.parse(data);
};

constsaveTodos = (todos) => {
fs.writeFileSync(todosFilePath, JSON.stringify(todos, null, 2));
};

app.get('/todos', (req, res) => {
consttodos = getTodos();
res.json(todos);
});

app.post('/todos', (req, res) => {
constnewTodo = req.body;
consttodos = getTodos();
todos.push(newTodo);
saveTodos(todos);
res.status(201).json(newTodo);
});

app.delete('/todos/:id', (req, res) => {
const{ id } = req.params;
lettodos = getTodos();
todos = todos.filter(todo => todo.id !== id);
saveTodos(todos);
res.status(204).send();
});
app.listen(PORT, () => {
console.log(`Server is running on http://localhost:${PORT}`);
});
```

Output:



- Need to solve leetcode probem Delete
- develop a communication skil Delete
- Going to theatre on 5pm today Delete

Software Requirements :

1. **Frontend**
   - React.js
   - Axios
   - Node.js (for development environment)
   - CSS (for styling)
2. **Backend**
   - Node.js
   - Express.js
   - cors, body-parser, and fs libraries
3. **Tools**
   - Code Editor (e.g., Visual Studio Code)
   - Node Package Manager (npm)
   - Browser (e.g., Chrome, Firefox)

Explanation :

1. **Frontend Functionality:**
    - The App component handles state management using React hooks.
    - The fetchTodos function fetches data from the backend using Axios.
    - The handleAddTodo function sends new todos to the backend and updates the UI.
    - The handleDeleteTodo function deletes todos from the backend and updates the UI.
2. **Backend Functionality:**
    - The backend handles requests via RESTful API endpoints:
        - ♣ GET /todos: Fetches all todos.
        - ♣ POST /todos: Adds a new todo.
        - ♣ DELETE /todos/:id: Deletes a todo by its ID.
    - Data is stored persistently in todos.json.
3. **Integration:**
    - The React frontend communicates with the Node.js backend via HTTP requests to manage todos.

Result:

Thus the program has been executed successfully and output verified.

Ex.No:9	**Create a simple Sign up and Login mechanism and authenticate the user using cookies. The user information can be stored in either MongoDB or MySQL. and the server should be built using NodeJS and Express Framework**
Date:	

Aim:

To build a simple user authentication system using Node.js, Express, MongoDB, and cookies for secure sign-up, login, and logout with JWT authentication.

Algorithm:

1. Start the server using Node.js and Express.
2. Connect to MongoDB for storing user data.
3. Serve index.html using Express.
4. Register User:
   - Hash password and store user in MongoDB.
   - Redirect to login page after successful signup.
5. Login User:
   - Verify credentials and generate a JWT token.
   - Store the token in cookies and show the dashboard.
6. Logout User:
   - Clear the authentication cookie and redirect to login.
7. Frontend UI:
   - Use fetch() for API requests.
   - Toggle between login, signup, and dashboard views dynamically.

Program:

Public/index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Auth System</title>
    <style>
        body { font-family: Arial, sans-serif; text-align: center; margin: 50px; }
        input { margin: 5px; padding: 10px; width: 200px; }
        button { padding: 10px 20px; cursor: pointer; }
        .hidden { display: none; }
    </style>
</head>
<body>
    <h1>Authentication System</h1>
```

```
<div id="nav">
    <button onclick="showPage('registerPage')">Sign Up</button>
    <button onclick="showPage('loginPage')">Login</button>
</div>

<div id="registerPage" class="hidden">
    <h2>Register</h2>
    <input type="text" id="regUsername" placeholder="Username" required>
    <input type="password" id="regPassword" placeholder="Password" required>
    <button id="registerBtn" onclick="register()">Sign Up</button>
</div>

<div id="loginPage">
    <h2>Login</h2>
    <input type="text" id="loginUsername" placeholder="Username" required>
    <input type="password" id="loginPassword" placeholder="Password" required>
    <button id="loginBtn" onclick="login()">Login</button>
</div>

<div id="dashboard" class="hidden">
    <h2>Welcome to Dashboard</h2>
    <button onclick="logout()">Logout</button>
</div>

<script>
    function showPage(page) {
        document.getElementById("registerPage").classList.add("hidden");
        document.getElementById("loginPage").classList.add("hidden");
        document.getElementById(page).classList.remove("hidden");
    }

    async function register() {
        const username = document.getElementById("regUsername").value;
        const password = document.getElementById("regPassword").value;
        constbtn = document.getElementById("registerBtn");

        btn.disabled = true; // Disable button while processing

        const res = await fetch("/register", {
            method: "POST",
            headers: { "Content-Type": "application/json" },
            body: JSON.stringify({ username, password })
        });
    const data = await res.json();
    alert(data.message);

    btn.disabled = false; // Re-enable button

    if (res.ok) {
        showPage('loginPage'); // Redirect to login page after signup
    }
}

async function login() {
    const username = document.getElementById("loginUsername").value;
    const password = document.getElementById("loginPassword").value;
    constbtn = document.getElementById("loginBtn");

    btn.disabled = true; // Disable button while processing

    const res = await fetch("/login", {
        method: "POST",
        headers: { "Content-Type": "application/json" },
        credentials: "include",
        body: JSON.stringify({ username, password })
    });
    const data = await res.json();

    btn.disabled = false; // Re-enable button

    if (res.ok) {
        document.getElementById("loginPage").classList.add("hidden");
        document.getElementById("dashboard").classList.remove("hidden");
        document.getElementById("nav").classList.add("hidden"); // Hide nav after login
    }
    alert(data.message);
}

async function logout() {
    await fetch("/logout", {
        method: "POST",
        credentials: "include"
    });
    document.getElementById("dashboard").classList.add("hidden");
    document.getElementById("nav").classList.remove("hidden"); // Show nav after logout
    showPage('loginPage'); // Redirect to login
    alert("Logged out successfully");
```

```
        }
    </script>
</body>
</html>
```

index.js

```
const express = require("express");
const mongoose = require("mongoose");
const path = require("path");
constbcrypt = require("bcryptjs");
constcookieParser = require("cookie-parser");
constjwt = require("jsonwebtoken");
constcors = require("cors");

const app = express();
const PORT = 5000;
const SECRET_KEY = "your_secret_key";

app.use(express.json());
app.use(cookieParser());
app.use(cors({ credentials: true, origin: "http://localhost:5000" }));

// Serve static frontend files
app.use(express.static(path.join(_dirname, "public")));

// Serve index.html as the default page
app.get("/", (req, res) => {
  res.sendFile(path.join(__dirname, "public", "index.html"));
});

// MongoDB Connection
mongoose.connect("mongodb+srv://1452dipakr:DIPAK1452@cluster0.w3mf3.mongodb.net/UserD
B?retryWrites=true&w=majority&appName=Cluster0", {
  useNewUrlParser: true,
  useUnifiedTopology: true,
}).then(() => console.log(" MongoDB Connected"))
  .catch(err =>console.error(" MongoDB Connection Error:", err));

// User Schema
constUserSchema = new mongoose.Schema({
  username: { type: String, required: true, unique: true },
  password: { type: String, required: true },
});
const User = mongoose.model("User", UserSchema);
```

```
// Register Route
app.post("/register", async (req, res) => {
  try {
    const{ username, password } = req.body;
    if (!username || !password) return res.status(400).json({ message: "All fields are required" });

    constexistingUser = await User.findOne({ username });
    if (existingUser) return res.status(400).json({ message: "User already exists" });

    consthashedPassword = await bcrypt.hash(password, 10);
    constnewUser = new User({ username, password: hashedPassword });
    await newUser.save();

    res.status(201).json({ message: "User registered successfully" });
  } catch (error) {
    res.status(500).json({ message: "Error registering user", error: error.message });
  }
});

// Login Route
app.post("/login", async (req, res) => {
  try {
    const{ username, password } = req.body;
    if (!username || !password) return res.status(400).json({ message: "All fields are required" });

    const user = await User.findOne({ username });
    if (!user) return res.status(400).json({ message: "Invalid username or password" });

    constisMatch = await bcrypt.compare(password, user.password);
    if (!isMatch) return res.status(400).json({ message: "Invalid username or password" });

    const token = jwt.sign({ userId: user._id }, SECRET_KEY, { expiresIn: "1h" });
    res.cookie("token", token, { httpOnly: true, sameSite: "Lax" }).json({ message: "Login
successful" });
  } catch (error) {
    res.status(500).json({ message: "Error logging in", error: error.message });
  }
});

// Logout Route
app.post("/logout", (req, res) => {
  res.clearCookie("token").json({ message: "Logged out successfully" });
});
```

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

Result:

Thus the program has been executed successfully and output verified.

Ex.No:10	Create and deploy a virtual machine using a virtual box that can be accessed from the host computer using SSH
Date:	

Aim

To create and deploy a virtual machine using VirtualBox, configure it to allow SSH access from the host computer, and test the SSH connection.

Algorithm

1. Install VirtualBox and Download Linux ISO
   a. Download and install VirtualBox from the official website.
   b. Download a Linux distribution ISO (e.g., Ubuntu) from the official site.
2. Create a Virtual Machine
   a. Open VirtualBox and click 'New'.
   b. Name the VM (e.g., LinuxVM), set the type to Linux, and version to Ubuntu (64-bit).
   c. Allocate memory (e.g., 2GB or 2048MB).
   d. Create a virtual hard disk (VDI format, dynamically allocated, size at least 10GB).
3. Attach the ISO File
   a. Go to Settings > Storage in VirtualBox.
   b. Under Controller: IDE, click the empty disk and choose the downloaded Linux ISO.
4. Configure Networking
   a. Go to Settings > Network.
   b. Set Adapter 1 to NAT for internet access.
   c. Enable Adapter 2 and set it to Host-Only Adapter to allow SSH connection from the host.
5. Start the VM and Install Linux
   a. Start the VM and follow the on-screen instructions to install the Linux OS.
   b. Set up the language, timezone, and disk configuration.
   c. Create a username and password.
6. Install OpenSSH Server on the VM
   a. Run the following commands to update and install OpenSSH:

```
sudo apt update
sudo apt install -y openssh-server
sudosystemctl enable ssh
sudosystemctl start ssh
sudosystemctl status ssh
```

7. Set Up Port Forwarding
   a. In VirtualBox, go to Settings > Network > Adapter 1.
   b. Click Advanced > Port Forwarding.
   c. Add a new rule:

   i. Name: ssh
   ii. Protocol: TCP
   iii. Host IP: 127.0.0.1
   iv. Host Port: 2222
   v. Guest IP: 10.0.2.15
   vi. Guest Port: 22

8. Test SSH Access from Host
   a. Find the IP address of the VM using:

```
ip a
```

   b. Connect to the VM using SSH from the host machine:

```
ssh username@127.0.0.1 -p 2222
```

Program (Bash Script to Enable SSH)

```
#!/bin/bash
# Update package list and install OpenSSH server
sudo apt update
sudo apt install -y openssh-server

# Enable SSH to start on boot
sudosystemctl enable ssh

# Start SSH service
sudosystemctl start ssh

# Check the status of SSH service
sudosystemctl status ssh
```

Expected Output

- On the VM: The SSH service should be running.
- On the Host: The SSH command should connect to the VM, prompting for the password. Once entered, terminal access to the VM should be established.

Result:

Thus the program has been executed successfully and output verified.

Ex.No:11	Create a Docker container that will deploy a Node.JS ping server using the Node.JS image
Date:	

AIM : The aim of this project is to create a simple HTTP server using Node.js that responds with "Hello World" to any incoming requests.

Prerequisites:

1. Ensure you have [Node.js](https://nodejs.org/) installed on your machine.

2. Install [Docker](https://www.docker.com/get-started) if you want to run the server in a Docker container.

1. Steps to Run in Docker:

Create a file named `Dockerfile` in the same directory as `server.js`. Use the following content for the Dockerfile:

```
  # Use the official Node.js image
  FROM node:14

  # Set the working directory
  WORKDIR /usr/src/app

  # Copy package.json and package-lock.json (if available)
  COPY package*.json ./

  # Install dependencies (if any)
  RUN npm install

  # Copy the rest of the application code
COPY . .
```

```
# Expose the port the app runs on
EXPOSE 3001

# Command to run the application
CMD ["node", "server.js"]
```

2. Build the Docker Image:

Open a terminal and navigate to the directory containing your `Dockerfile` and `server.js`. Run the following command:

```
docker build -t node-ping-server .
```

3. Run the Docker Container:

After the image is built, run the container with:

```
docker run -p 3001:3001 node-ping-server
```

4. Access the Server:

Open your web browser and navigate to `**http: //localhost:3001**`. You should see the response:

```
Hello World
```

SERVER.JS

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

const hostname = "0.0.0.0"; // Listen on all interfaces
const port = 3001;

const server = http.createServer((req, res) => {
res.statusCode = 200;
res.setHeader("Content-Type", "text/plain");
res.end("Hello World\n");
});

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

DOCKERFILE

```
# Use the official Node.js image
FROM node:14

# Set the working directory
WORKDIR /usr/src/app

# Copy package.json and package-lock.json (if available)
COPY package*.json ./

# Install dependencies (if any)
RUN npm install

# Copy the rest of the application code
COPY . .

# Expose the port the app runs on
EXPOSE 3001

# Command to run the application
```

```
CMD ["node", "server.js"]
```

Result

This simple Node.js ping server can be used for testing and monitoring purposes. Also we can extend its functionality as needed for more complex app