Experiment - 1

AIM:

To write an HTML program that creates an ordered list, unordered list, a table, and incorporates multimedia elements like an image, audio, and video using only HTML

OBJECTIVE:

- Learn how to structure content using basic HTML tags.
- Understand how to embed multimedia (images, audio, video) into a web page.
- Gain hands-on experience in using HTML5 features.

Source code: index.html Program

```
<!DOCTYPE html>
<html>
<head>
  <title>HTML Lists, Table, Image, Audio, and Video</title>
</head>
<body>

  <h1>HTML Experiment: Lists, Table, Image, Audio & Video</h1>


  <h2>Unordered List of Programming Languages</h2>
  <ul>
    <li>Python</li>
    <li>Java</li>
    <li>C++</li>
    <li>HTML</li>
    <li>JavaScript</li>
  </ul>

  <h2>Ordered List: Steps to Build a Web Page</h2>
  <ol>
    <li>Create HTML File</li>
    <li>Add Content Using Tags</li>
    <li>Insert Media Elements</li>
    <li>Save and Open in Browser</li>
  </ol>
  <h2>Student Details Table</h2>
  <table border="1">
    <tr>
      <th>Roll No</th>
      <th>Name</th>
```

```
            <th>Subject</th>
            <th>Marks</th>
        </tr>
        <tr>
            <td>101</td>
            <td>Rahul</td>
            <td>Math</td>
            <td>85</td>
        </tr>
        <tr>
            <td>102</td>
            <td>Sneha</td>
            <td>Science</td>
            <td>90</td>
        </tr>
        <tr>
            <td>103</td>
            <td>Amit</td>
            <td>English</td>
            <td>78</td>
        </tr>
    </table>

    <h2>Sample Image</h2>
    <img src="https://via.placeholder.com/300x200" alt="Sample Image">

    <h2>Sample Audio</h2>
    <audio controls>
        <source src="sample-audio.mp3" type="audio/mpeg">
        Your browser does not support the audio tag.
    </audio>

    <h2>Sample Video</h2>
    <video width="320" height="240" controls>
        <source src="sample-video.mp4" type="video/mp4">
        Your browser does not support the video tag.
    </video>

</body>
</html>
```

STEP-BY-STEP INSTRUCTIONS:

1. Create a new folder and name it html-experiment.
2. Create an HTML file named index.html.
3. Copy and paste the above HTML code into index.html.
4. Place sample media files:
    - Download or use your own sample-audio.mp3 and sample-video.mp4 files.
    - Save them in the same folder as your index.html file.
5. Open the HTML file in a browser by double-clicking or right-click → Open With → Choose Browser.

Result:

- An HTML page displaying:
    - An **unordered list** of programming languages.
    - An **ordered list** describing steps to build a webpage.
    - A **table** containing student details.
    - An embedded **image**.
    - A **playable audio** clip.
    - A **playable video** clip.

Experiment - 2

AIM:

To design and develop a simple personal portfolio web page using only HTML and CSS.

OBJECTIVE:

- Understand the structure of a basic HTML document.
- Learn how to use CSS for styling web pages.
- Create a sample application (Portfolio Page) showcasing user details.

Source code: index.html Program

```
<!DOCTYPE html>
<html lang="en">
<head>
   <meta charset="UTF-8">
   <meta name="viewport" content="width=device-width, initial-scale=1.0">
   <title>My Portfolio</title>
   <link rel="stylesheet" href="style.css">
</head>
<body>

   <header>
      <h1>John Doe</h1>
      <p>Web Developer | Designer | Programmer</p>
   </header>

   <nav>
      <a href="#">Home</a>
      <a href="#">About</a>
      <a href="#">Projects</a>
      <a href="#">Contact</a>
   </nav>
  <section class="about">
      <h2>About Me</h2>
      <p>I am a passionate web developer with experience in creating responsive and user-friendly websites. I love learning new technologies and building innovative web applications</p>
   </section>
```

```
    <section class="projects">
        <h2>Projects</h2>
        <ul>
            <li>Portfolio Website</li>
            <li>Todo List App</li>
            <li>Weather App</li>
        </ul>
    </section>

    <footer>
        <p>Contact me at: johndoe@example.com</p>
        <p>© 2025 John Doe</p>
    </footer>

</body>
</html>
```

Source code of ***style.css*** external sheet:-

```
/* Reset default browser styles */
* {
    margin: 0;
    padding: 0;
    box-sizing: border-box;
}

body {
    font-family: Arial, sans-serif;
    line-height: 1.6;
    background-color: #f4f4f4;
    color: #333;
    padding: 20px;
}

header {
    text-align: center;
    padding: 20px;
    background-color: #333;
    color: white;
}
```

```
nav {
    display: flex;
    justify-content: center;
    background-color: #444;
    margin-top: 10px;
}

nav a {
    color: white;
    padding: 14px 20px;
    text-decoration: none;
    display: inline-block;
}

nav a:hover {
    background-color: #666;
}

.about, .projects {
    background: white;
    margin: 20px auto;
    padding: 20px;
    max-width: 800px;
    border-radius: 8px;
    box-shadow: 0 2px 5px rgba(0,0,0,0.1);
}

footer {
    text-align: center;
    padding: 10px;
    margin-top: 20px;
    background-color: #333;
    color: white;
}
```

STEP-BY-STEP INSTRUCTIONS:

1. Create Project Folder

- Create a new folder named portfolio-site.

2. Create HTML File

- Inside the folder, create a file named index.html.
- Copy and paste the HTML code provided above.

3. Create CSS File

- In the same folder, create another file named style.css.
- Copy and paste the CSS code provided above.

4. Open the Web Page

- Double-click on index.html or right-click and select **"Open with browser"**.
- Your portfolio webpage should now be visible.

Result:

A clean, responsive personal portfolio website with sections like About Me, Projects, and Contact Info, styled using CSS.

Experiment - 3

AIM:

To develop a simple client-server application using Node.js and Express.js that handles basic HTTP GET and POST requests.

OBJECTIVE:

- Understand the working of Node.js as a backend runtime environment.
- Learn to use Express.js for setting up a basic web server.
- Handle routing for client-server communication.

TOOLS & TECHNOLOGIES USED:

- Node.js
- Express.js
- Code Editor (VS Code, Sublime, etc.)
- Terminal or Command Prompt
- Web Browser (for testing)

STEP-BY-STEP PROCEDURE

1. Install Node.js

- Download and install from https://nodejs.org
- Verify installation with:

Command

```
node -v
npm -v
```

2. Create a Project Folder

Command

```
mkdir simple-server
cd simple-server
```

3. Initialize the Node.js Project

```bash
npm init -y
```

4. Install Express.js

```bash
npm install express
```

5. Create the Server File

- Create a file named server.js inside the project folder.

Source code:-

```
const express = require('express');
const app = express();
const port = 3000;

app.use(express.json());

// GET route
app.get('/', (req, res) => {
  res.send('Welcome to the Simple Node.js + Express Server!');
});

// POST route
app.post('/greet', (req, res) => {
  const name = req.body.name;
  res.send(`Hello, ${name}! Welcome to the server.`);
});

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

6. Run the Server

```bash
node server.js
```

You should see:

```arduino
Server running at http://localhost:3000
```

7. Test in Browser or Postman

- Open http://localhost:3000/ in your browser → You'll see the welcome message.
- Use **Postman** or **cURL** to test the POST request:
  - URL: http://localhost:3000/greet
  - Method: POST
  - Body (JSON):

```
{
  "name": "Alice"
}
```

Response:

```pgsql
Hello, Alice! Welcome to the server.
```

OUTPUT:

1. **GET Request Response:**
2.

```
Welcome to the Simple Node.js + Express Server!
```

3. **POST Request Response:**

```
Hello, Alice! Welcome to the server.
```

RESULT:

A simple client-server application was successfully developed using Node.js and Express.js. The server handled both GET and POST requests, demonstrating how web servers and clients can communicate using HTTP methods.

Experiment - 4

AIM

To develop a dynamic Single Page Application (SPA) using React.js that allows users to navigate between multiple views (pages) such as Home, About, and Users, including dynamic routing for user profiles — all without reloading the browser.

OUTCOME

A fully functional and dynamic SPA built with React.js, featuring:

- Client-side routing with react-router-dom
- Dynamic route handling using URL parameters
- Seamless navigation between pages without full-page reloads
- Component-based structure following React principles

Step 1: Set Up the React App

npx create-react-app my-spa
cd my-spa
npm install react-router-dom

What happens here?

- create-react-app sets up a React project with build tools.
- react-router-dom is installed for routing.

Step 2: Configure the Router

File: index.js

```
import React from 'react';
import ReactDOM from 'react-dom/client';
import { BrowserRouter } from 'react-router-dom';
import App from './App';

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(
  <BrowserRouter>
    <App />
  </BrowserRouter>
);
```

Step 3: **Create Main Navigation and Route Setup**

```
import React from 'react';
```

```
import { Routes, Route } from 'react-router-dom';
import Navbar from './components/Navbar';
import Home from './components/Home';
import About from './components/About';
import Users from './components/Users';
import UserProfile from './components/UserProfile';

function App() {
  return (
    <>
      <Navbar />
      <Routes>
        <Route path="/" element={<Home />} />
        <Route path="/about" element={<About />} />
        <Route path="/users" element={<Users />} />
        <Route path="/users/:id" element={<UserProfile />} />
      </Routes>
    </>
  );
}

export default App;
```

Step 4: **Build Reusable Components**

```
import React from 'react';
import { Link } from 'react-router-dom';

const Navbar = () => (
  <nav>
    <Link to="/">Home</Link> |
    <Link to="/about">About</Link> |
    <Link to="/users">Users</Link>
  </nav>
);

export default Navbar;
const Home = () => (
  <div>
    <h2>Home</h2>
    <p>Welcome to the Home Page.</p>
  </div>
);
export default Home;
```

```
 const About = () => (
  <div>
    <h2>About</h2>
    <p>This is a SPA built using React.js.</p>
  </div>
);

export default About;
```

- Users.js

```
import React from 'react';
import { Link } from 'react-router-dom';

const users = [
  { id: 1, name: "Alice" },
  { id: 2, name: "Bob" },
];

const Users = () => (
  <div>
    <h2>Users</h2>
    <ul>
      {users.map(user => (
        <li key={user.id}>
          <Link to={`/users/${user.id}`}>{user.name}</Link>
        </li>
      ))}
    </ul>
  </div>
);

export default Users;
```

- UserProfile.js

```
import React from 'react';
import { useParams } from 'react-router-dom';

const UserProfile = () => {
  const { id } = useParams();

  return (
    <div>
```

```
      <h2>User Profile</h2>
      <p>Viewing profile for user with ID: {id}</p>
    </div>
  );
};

export default UserProfile;
```

Step 5: Style the App (Optional)
File: App.css

```
nav {
  background-color: #333;
  padding: 10px;
}

nav a {
  color: white;
  margin-right: 10px;
  text-decoration: none;
}

body {
  font-family: Arial, sans-serif;
  margin: 0;
  padding: 20px;
}
```

- **Step 6: Run the Application**

npm start
Open your browser at **http://localhost:3000**.

 Click through links — no page reload.
 Dynamic user profiles open based on route.

Final Output
A functional, component-based React.js SPA with:

- Navigation between multiple routes
- Dynamic user profile pages
- No page reloads
- Clean, structured React code

Experiment - 5

AIM

To build a To-Do list application using React.js and Redux to manage the global state, allowing users to add, delete, and toggle the completion status of tasks.

OUTCOME

A fully functional To-Do list app that:

- Uses **React.js** for UI rendering
- Uses **Redux** for centralized state management
- Supports adding, removing, and toggling tasks
- Demonstrates a clean architectural separation of concerns

Step 1: Create React App & Install Redux Packages

npx create-react-app redux-todo-app
cd redux-todo-app
npm install @reduxjs/toolkit react-redux

Step 2: **Project Structure**

```
redux-todo-app/
├── src/
│   ├── app/
│   │   └── store.js
│   ├── features/
│   │   └── todos/
│   │       ├── todosSlice.js
│   │       └── TodoList.js
│   ├── App.js
│   └── index.js
```

Step 3: **Setup Redux Store**

src/app/store.js

```
import { configureStore } from '@reduxjs/toolkit';
import todosReducer from '../features/todos/todosSlice';

export const store = configureStore({
  reducer: {
    todos: todosReducer,
  },
});
```

Step 4: **Define Redux Slice**

```
import { createSlice } from '@reduxjs/toolkit';

let nextId = 1;

const todosSlice = createSlice({
  name: 'todos',
  initialState: [],
  reducers: {
    addTodo: (state, action) => {
      state.push({
        id: nextId++,
        text: action.payload,
        completed: false
      });
    },
    toggleTodo: (state, action) => {
      const todo = state.find(todo => todo.id === action.payload);
      if (todo) {
        todo.completed = !todo.completed;
      }
    },
    deleteTodo: (state, action) => {
      return state.filter(todo => todo.id !== action.payload);
    }
  }
});

export const { addTodo, toggleTodo, deleteTodo } = todosSlice.actions;
export default todosSlice.reducer;
```

Step 5: Integrate Redux Store with React App

```
import React from 'react';
import ReactDOM from 'react-dom/client';
import { Provider } from 'react-redux';
import { store } from './app/store';
import App from './App';

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(
  <Provider store={store}>
    <App />
  </Provider>
```

);

Step 6: **Build the To-Do UI**

```
import React, { useState } from 'react';
import { useSelector, useDispatch } from 'react-redux';
import { addTodo, toggleTodo, deleteTodo } from './todosSlice';

const TodoList = () => {
  const [text, setText] = useState('');
  const todos = useSelector(state => state.todos);
  const dispatch = useDispatch();

  const handleAdd = () => {
    if (text.trim()) {
      dispatch(addTodo(text));
      setText('');
    }
  };

  return (
    <div>
      <h2>To-Do List</h2>
      <input
        type="text"
        value={text}
        onChange={e => setText(e.target.value)}
        placeholder="Enter task"
      />
      <button onClick={handleAdd}>Add</button>

      <ul>
        {todos.map(todo => (
          <li key={todo.id} style={{ marginTop: '10px' }}>
            <span
              onClick={() => dispatch(toggleTodo(todo.id))}
              style={{
                textDecoration: todo.completed ? 'line-through' : 'none',
                cursor: 'pointer'
              }}
            >
              {todo.text}
            </span>
            <button
              onClick={() => dispatch(deleteTodo(todo.id))}
```

```
            style={{ marginLeft: '10px' }}
          >
            Delete
          </button>
        </li>
      ))}
    </ul>
  </div>
 );
};

export default To do List;
```

Step 7: Render in App

src/App.js

```
import React from 'react';
import TodoList from './features/todos/TodoList';

function App() {
 return (
  <div style={{ padding: '20px' }}>
   <TodoList />
  </div>
 );
}

export default App;
```

Final Output(result)

You now have a **React + Redux** based To-Do list application that:

- Adds tasks
- Toggles tasks (complete/incomplete)
- Deletes tasks
- Manages state globally via Redux

Experiment - 6

AIM

To create a user-friendly interactive form using HTML, CSS3, and JavaScript that validates input fields in real-time (e.g., name, email, password), providing immediate feedback for errors.

OUTCOME

A responsive and styled form that:

- Validates user input on blur and submit
- Highlights errors with real-time feedback
- Prevents submission if validation fails
- Uses JavaScript for logic and CSS3 for visual cues

Technologies Used

- HTML5
- CSS3
- Vanilla JavaScript

Write a source code:

1. HTML (index.html)

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Interactive Form</title>
  <link rel="stylesheet" href="style.css">
</head>
<body>
  <div class="form-container">
    <h2>Registration Form</h2>
    <form id="registrationForm" novalidate>
      <div class="form-group">
        <label for="name">Full Name:</label>
        <input type="text" id="name" placeholder="Enter full name" required>
        <small class="error-message"></small>
      </div>

      <div class="form-group">
        <label for="email">Email Address:</label>
        <input type="email" id="email" placeholder="Enter email" required>
        <small class="error-message"></small>
      </div>
```

```
      <div class="form-group">
        <label for="password">Password:</label>
        <input type="password" id="password" placeholder="Minimum 6 characters" required>
        <small class="error-message"></small>
      </div>

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

  <script src="script.js"></script>
</body>
</html>
```

2. CSS (style.css)

```
body {
  font-family: 'Segoe UI', sans-serif;
  background: #f2f2f2;
  margin: 0;
  padding: 0;
}

.form-container {
  max-width: 400px;
  background: white;
  margin: 80px auto;
  padding: 30px;
  border-radius: 8px;
  box-shadow: 0 0 10px rgba(0, 0, 0, 0.1);
}

h2 {
  text-align: center;
  margin-bottom: 20px;
}

.form-group {
  margin-bottom: 20px;
  position: relative;
}

label {
  display: block;
  margin-bottom: 6px;
}

input {
  width: 100%;
  padding: 10px;
  border: 2px solid #ccc;
```

```
  border-radius: 5px;
  font-size: 16px;
  transition: border-color 0.3s;
}

input:focus {
  border-color: #007BFF;
  outline: none;
}

input.error {
  border-color: red;
}

.error-message {
  color: red;
  font-size: 13px;
  margin-top: 5px;
  display: none;
}

input.error + .error-message {
  display: block;
}

button {
  width: 100%;
  padding: 12px;
  font-size: 16px;
  background: #007BFF;
  color: white;
  border: none;
  border-radius: 5px;
  cursor: pointer;
}

button:hover {
  background: #0056b3;
}
```

4.JavaScript (script.js)

```
document.addEventListener("DOMContentLoaded", () => {
  const form = document.getElementById("registrationForm");
  const nameInput = document.getElementById("name");
  const emailInput = document.getElementById("email");
  const passwordInput = document.getElementById("password");

  function showError(input, message) {
    const formGroup = input.parentElement;
    const errorMsg = formGroup.querySelector(".error-message");
```

```
  input.classList.add("error");
  errorMsg.textContent = message;
  errorMsg.style.display = "block";
}

function clearError(input) {
  const formGroup = input.parentElement;
  const errorMsg = formGroup.querySelector(".error-message");

  input.classList.remove("error");
  errorMsg.textContent = "";
  errorMsg.style.display = "none";
}

function validateName() {
  const name = nameInput.value.trim();
  if (name === "") {
    showError(nameInput, "Name is required.");
    return false;
  }
  clearError(nameInput);
  return true;
}

function validateEmail() {
  const email = emailInput.value.trim();
  const regex = /^[^\s@]+@[^\s@]+\.[^\s@]+$/;

  if (email === "") {
    showError(emailInput, "Email is required.");
    return false;
  } else if (!regex.test(email)) {
    showError(emailInput, "Enter a valid email.");
    return false;
  }

  clearError(emailInput);
  return true;
}

function validatePassword() {
  const password = passwordInput.value.trim();

  if (password === "") {
    showError(passwordInput, "Password is required.");
    return false;
  } else if (password.length < 6) {
    showError(passwordInput, "Password must be at least 6 characters.");
    return false;
  }

  clearError(passwordInput);
  return true;
}
```

```
  // Real-time validation
  nameInput.addEventListener("blur", validateName);
  emailInput.addEventListener("blur", validateEmail);
  passwordInput.addEventListener("blur", validatePassword);

  form.addEventListener("submit", function (e) {
    e.preventDefault();

    const isNameValid = validateName();
    const isEmailValid = validateEmail();
    const isPasswordValid = validatePassword();

    if (isNameValid && isEmailValid && isPasswordValid) {
      alert("Form submitted successfully!");
      form.reset();
    }
  });
});
```

Result: A form will show on browser, responsive form

Experiment - 7

AIM

To develop a RESTful API using Node.js and Express.js that supports basic Create, Read, Update, and Delete (CRUD) operations.

OUTCOME

A working REST API with the following endpoints:

- GET /users - Get all users
- GET /users/:id - Get a single user by ID
- POST /users - Create a new user
- PUT /users/:id - Update a user
- DELETE /users/:id - Delete a user

Step 1: Initialize Node.js Project

```
mkdir rest-api
cd rest-api
npm init -y
```

Step 2: Install Express

```
npm install express
```

Step 3: Project Structure

```
rest-api/
├── index.js        # Main server file
└── data.js         # Sample user data (in-memory)
```

Step 4: Create Sample User Data (data.js)

```
let users = [
  { id: 1, name: "Alice", email: "alice@example.com" },
  { id: 2, name: "Bob", email: "bob@example.com" }
];

module.exports = users;
```

Step 5: Create API in index.js

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

let users = require('./data');

// Middleware to parse JSON body
app.use(express.json());

// 📍 GET /users - Get all users
app.get('/users', (req, res) => {
  res.json(users);
});
```

```
// 📍 GET /users/:id - Get user by ID
app.get('/users/:id', (req, res) => {
  const user = users.find(u => u.id === parseInt(req.params.id));
  if (!user) return res.status(404).json({ message: "User not found" });
  res.json(user);
});

// 📍 POST /users - Create a new user
app.post('/users', (req, res) => {
  const { name, email } = req.body;
  const newUser = {
    id: users.length + 1,
    name,
    email
  };
  users.push(newUser);
  res.status(201).json(newUser);
});

// 📍 PUT /users/:id - Update a user
app.put('/users/:id', (req, res) => {
  const { name, email } = req.body;
  const user = users.find(u => u.id === parseInt(req.params.id));
  if (!user) return res.status(404).json({ message: "User not found" });

  user.name = name || user.name;
  user.email = email || user.email;

  res.json(user);
});

// 📍 DELETE /users/:id - Delete a user
app.delete('/users/:id', (req, res) => {
  const id = parseInt(req.params.id);
  const index = users.findIndex(u => u.id === id);
  if (index === -1) return res.status(404).json({ message: "User not found" });

  users.splice(index, 1);
  res.status(204).send();
});

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

Step 6: Run the Server

```
node index.js
```

Test Your API with Tools Like:

- Postman (GUI-based)
- curl (Command-line)

- Thunder Client (VSCode Extension)

Example API Requests
POST /users

```
{
  "name": "Charlie",
  "email": "charlie@example.com"
}
```

GET /users
- Returns all users.

GET /users/1
- Returns user with ID = 1.

PUT /users/1

```
{
  "name": "Alice Updated"
}
```

Final Output(Result)
You now have a complete, running **RESTful API using Node.js and Express.js** supporting full CRUD operations.

Experiment – 8

AIM

To implement JWT-based authentication in a Node.js + Express.js backend, including user login, token generation, protected routes, and token verification.

OUTCOME

A backend server with:

- /login – for generating JWT tokens after verifying credentials
- /protected – a route accessible only with a valid JWT
- Middleware to verify JWT in request headers

Step 1: Initialize Project

```
mkdir jwt-auth-api
cd jwt-auth-api
npm init -y
npm install express jsonwebtoken bcryptjs
```

Step 2: Project Structure

```
jwt-auth-api/
├── index.js          # Entry point
└── users.js          # Fake user DB (in-memory)
```

Step 3: Create a Dummy User DB (users.js)

```
const bcrypt = require('bcryptjs');

const users = [
  {
    id: 1,
    username: 'admin',
    password: bcrypt.hashSync('password123', 10)  // Store hashed password
  }
];

module.exports = users;
```

Step 4: Build Server with JWT Logic (index.js)

```
const express = require('express');
const jwt = require('jsonwebtoken');
const bcrypt = require('bcryptjs');
const users = require('./users');

const app = express();
const PORT = 3000;
const SECRET_KEY = 'mysecretkey'; // Store in env file for real apps

app.use(express.json());

// Login Endpoint
app.post('/login', (req, res) => {
```

```
  const { username, password } = req.body;

  const user = users.find(u => u.username === username);
  if (!user) return res.status(401).json({ message: 'Invalid username' });

  const isMatch = bcrypt.compareSync(password, user.password);
  if (!isMatch) return res.status(401).json({ message: 'Invalid password' });

  const token = jwt.sign({ id: user.id, username: user.username }, SECRET_KEY, {
    expiresIn: '1h'
  });

  res.json({ token });
});

// Middleware to verify JWT
function authenticateToken(req, res, next) {
  const authHeader = req.headers['authorization']; // Format: Bearer <token>
  const token = authHeader && authHeader.split(' ')[1];

  if (!token) return res.status(401).json({ message: 'Access token missing' });

  jwt.verify(token, SECRET_KEY, (err, user) => {
    if (err) return res.status(403).json({ message: 'Invalid or expired token' });
    req.user = user;
    next();
  });
}

// Protected Route
app.get('/protected', authenticateToken, (req, res) => {
  res.json({
    message: 'This is a protected route!',
    user: req.user
  });
});

// Public Route
app.get('/', (req, res) => {
  res.send('JWT Auth API is running');
});

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

JWT Flow

1. **Client sends credentials** to /login.
2. If valid, server responds with a **JWT token**.
3. Client includes JWT in **Authorization header** like:
   Authorization: Bearer <your-token-here>
4. Server uses middleware to **verify token** before allowing access to protected routes.

Test with Postman or curl

POST /login
Body:
```
{
  "username": "admin",
  "password": "password123"
}
```

Response:
```
{
  "token": "<JWT_TOKEN>"
}
```

GET /protected
Header:
Authorization: Bearer <JWT_TOKEN>

Response:
```
{
  "message": "This is a protected route!",
  "user": {
    "id": 1,
    "username": "admin",
    "iat": ...,
    "exp": ...
  }
}
```

Final Output
You now have:

- A working **JWT-based login system**
- A **protected API endpoint**
- Middleware that verifies token validity

Experiment - 9

AIM

To develop a real-time chat application using Node.js and WebSockets that allows multiple users to send and receive messages instantly in a chat room environment**.**

Technology Stack

- **Node.js** — server environment
- **Socket.IO** — simplifies WebSocket implementation
- **Express.js** — for serving frontend
- **HTML + CSS + JS** — for basic frontend

Folder Structure

realtime-chat-app/
├── server.js
├── public/
│ ├── index.html
│ └── style.css

- **Step 1: Initialize Project**

```
mkdir realtime-chat-app
cd realtime-chat-app
npm init -y
npm install express socket.io
```

- **Step 2: Create server.js**

```
const express = require('express');
const http = require('http');
const { Server } = require('socket.io');

const app = express();
const server = http.createServer(app);
const io = new Server(server);

// Serve static files from "public"
app.use(express.static('public'));

// WebSocket logic
io.on('connection', (socket) => {
  console.log('✅ New user connected:', socket.id);

  // When user sends a message
  socket.on('chat message', (msg) => {
    io.emit('chat message', msg); // broadcast to all clients
  });

  socket.on('disconnect', () => {
    console.log('❌ User disconnected:', socket.id);
```

```
  });
});

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

- **Step 3: Create public/index.html**

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Real-Time Chat</title>
  <link rel="stylesheet" href="style.css">
</head>
<body>
  <div class="chat-container">
    <h2>💬 WebSocket Chat</h2>
    <ul id="messages"></ul>
    <form id="chat-form">
      <input id="message" autocomplete="off" placeholder="Type your message..." />
      <button>Send</button>
    </form>
  </div>

  <script src="/socket.io/socket.io.js"></script>
  <script>
    const socket = io();

    const form = document.getElementById('chat-form');
    const input = document.getElementById('message');
    const messages = document.getElementById('messages');

    form.addEventListener('submit', (e) => {
      e.preventDefault();
      if (input.value) {
        socket.emit('chat message', input.value);
        input.value = '';
      }
    });

    socket.on('chat message', (msg) => {
      const item = document.createElement('li');
      item.textContent = msg;
      messages.appendChild(item);
      window.scrollTo(0, document.body.scrollHeight);
    });
  </script>
</body>
</html>
```

- **Step 4: Create public/style.css (Optional)**

```
body {
  font-family: Arial, sans-serif;
  background: #f0f0f0;
  padding: 0;
  margin: 0;
}

.chat-container {
  width: 400px;
  margin: 50px auto;
  background: white;
  padding: 20px;
  border-radius: 8px;
  box-shadow: 0 0 10px rgba(0, 0, 0, 0.1);
}

#messages {
  list-style-type: none;
  padding: 0;
  max-height: 300px;
  overflow-y: auto;
}

#messages li {
  padding: 8px;
  border-bottom: 1px solid #eee;
}

form {
  display: flex;
  margin-top: 10px;
}

input {
  flex: 1;
  padding: 10px;
  font-size: 16px;
}

button {
  padding: 10px 15px;
  font-size: 16px;
  background: #007BFF;
  color: white;
  border: none;
}
```

- **Step 5: Run Your Server**

```
node server.js
```

Now open your browser and go to:

- http://localhost:3000

Open multiple tabs to simulate multiple users. Type messages and see them broadcast in real time!

OUTPUT: shown on browser, Rendering on web page

Experiment – 10

AIM

To develop a Progressive Web App (PWA) that can function offline using service workers, and demonstrate features like installability, responsiveness, and caching of static assets.

Tools Required

- Code editor (e.g., VS Code)
- Local server (e.g., Live Server, or http-server)
- Modern browser (e.g., Chrome)

Step 1: Create Basic Web App Structure

```
/pwa-offline-app/
│
├── index.html
├── styles.css
├── app.js
├── manifest.json
└── service-worker.js
```

index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>PWA Offline Demo</title>
  <link rel="stylesheet" href="styles.css">
  <link rel="manifest" href="manifest.json">
</head>
<body>
  <h1>Progressive Web App</h1>
  <p>This app works offline!</p>
  <script src="app.js"></script>
</body>
</html>
```

styles.css

```
body {
  font-family: sans-serif;
  text-align: center;
  margin-top: 50px;
}
```

app.js (Register Service Worker)

```
if ('serviceWorker' in navigator) {
  window.addEventListener('load', () => {
    navigator.serviceWorker.register('service-worker.js')
      .then(reg => console.log('Service Worker registered:', reg.scope))
      .catch(err => console.log('Service Worker registration failed:', err));
  });
}
```

manifest.json (Web App Manifest)

```
{
  "name": "PWA Offline App",
  "short_name": "PWAOffline",
  "start_url": "./index.html",
  "display": "standalone",
  "background_color": "#ffffff",
  "theme_color": "#000000",
  "icons": [
    {
      "src": "https://via.placeholder.com/192",
      "sizes": "192x192",
      "type": "image/png"
    },
    {
      "src": "https://via.placeholder.com/512",
      "sizes": "512x512",
      "type": "image/png"
    }
  ]
}
```

service-worker.js (Basic Caching for Offline)

```
const CACHE_NAME = 'pwa-offline-cache-v1';
const FILES_TO_CACHE = [
  '/',
  '/index.html',
  '/styles.css',
  '/app.js'
];

// Install event - caching static assets
self.addEventListener('install', event => {
  event.waitUntil(
    caches.open(CACHE_NAME)
      .then(cache => cache.addAll(FILES_TO_CACHE))
  );
});

// Activate event
```

```
self.addEventListener('activate', event => {
  event.waitUntil(
    caches.keys().then(keyList => {
      return Promise.all(
        keyList.map(key => {
          if (key !== CACHE_NAME) return caches.delete(key);
        })
      );
    })
  );
  return self.clients.claim();
});

// Fetch event - serving cached content
self.addEventListener('fetch', event => {
  event.respondWith(
    caches.match(event.request)
      .then(response => response || fetch(event.request))
  );
});
```

- **Step 2: Serve Your App via HTTPS or Localhost**

You can use a simple local server like:

- VS Code extension: **Live Server**
- Or use terminal:

```
npx http-server -c-1
```

- **Step 3: Test PWA Features**

1. Open in Chrome and inspect → Application tab.
2. Check if:
    - Manifest is loaded
    - Service worker is registered
    - App is Installable
3. Go offline and reload – App should still work!

Result

The developed PWA demonstrates:

- Service worker caching for offline access.
- Manifest for installability and app-like behavior.
- Responsive, fast, and functional UI even when offline.

Experiment – 11

AIM

To deploy a simple Node.js backend application using **serverless architecture** with **AWS Lambda** or **Firebase Functions**, demonstrating how backend logic can run in the cloud without managing servers.

Requirements

- AWS account
- Node.js installed
- AWS CLI configured
- serverless framework (optional but recommended)

➢ **Step 1: Install Serverless Framework (optional but simplifies deployment)**

```
npm install -g serverless
```

➢ **Step 2: Create a Simple Node.js Function**

```
serverless create --template aws-nodejs --path aws-lambda-app
cd aws-lambda-app
```

```
This creates:
/aws-lambda-app/
├── handler.js
└── serverless.yml
```

➢ **Step 3: Update handler.js**

```
// handler.js
module.exports.hello = async (event) => {
 return {
  statusCode: 200,
  body: JSON.stringify({ message: 'Hello from AWS Lambda!' }),
 };
};
```

➢ **Step 4: Update serverless.yml**

```
service: aws-lambda-app

provider:
 name: aws
 runtime: nodejs18.x
 region: us-east-1

functions:
 hello:
  handler: handler.hello
  events:
```

```
    - http:
        path: hello
        method: get
```

- **Step 5: Deploy to AWS**

```
serverless deploy
```

After deployment, you'll get a URL like:

https://xxxx.execute-api.us-east-1.amazonaws.com/dev/hello

Test the Endpoint
Open the URL in your browser or use:

```
curl https://xxxx.execute-api.us-east-1.amazonaws.com/dev/hello
```

RESULT (AWS Lambda)

A Node.js function is now running in the cloud and can be accessed via an HTTP endpoint using AWS Lambda and API Gateway.

Experiment – 12

AIM

To optimize a web application for better performance by applying key techniques like **code minification**, **lazy loading**, **image optimization**, and **caching strategies**, resulting in faster load times and better user experience.

Step 1: Create Basic Web App Structure

/optimized-web-app/
├── index.html
├── style.css
├── script.js
├── image.jpg

1. Minify HTML, CSS, and JS

Tools

- HTMLMinifier
- CSSNano
- Terser (for JS)

Before Minification

```
<!-- index.html -->
<html>
  <head>
    <link rel="stylesheet" href="style.css">
  </head>
  <body>
    <h1>Hello</h1>
    <script src="script.js"></script>
  </body>
</html>
```

After Minification

```
<html><head><link rel="stylesheet" href="style.css"></head><body><h1>Hello</h1><script src="script.js"></script></body></html>
```

You can automate this using tools like **Webpack**, **Gulp**, or **Vite**.

2. Optimize Ima

```
<!-- Before -->
<img src="image.jpg" alt="Banner">

<!-- After -->
<img src="image.webp" alt="Banner" loading="lazy" width="600" height="400">
```

3. Implement Lazy Loading
Lazy loading defers the loading of off-screen content.

For Images
<img src="image.webp" loading="lazy" alt="Lazy Image">

For JavaScript Modules (Dynamic Import)

```
// script.js
document.getElementById('loadBtn').addEventListener('click', () => {
  import('./heavyComponent.js').then(module => {
    module.loadComponent();
  });
});
```

4. Bundle & Tree Shake JS
Use Webpack or Vite to:

- Bundle files into one.
- Remove unused code (tree shaking).
- Minify automatically.

```
npm init -y
npm install vite
```

Add this to package.json scripts:

```
"scripts": {
  "dev": "vite",
  "build": "vite build"
}
```

5. Remove Unused CSS (PurgeCSS)
Unused CSS slows loading.
Install PurgeCSS:

```
npm install @fullhuman/postcss-purgecss
```

Then configure to remove unused styles during the build.

6. Use HTTP Caching
In your server config (e.g., Apache, Nginx, or vite.config.js), enable long-term caching:

```
// vite.config.js example
export default {
  build: {
    rollupOptions: {
      output: {
```

```
      entryFileNames: 'assets/[name].[hash].js'
    }
  }
 }
}
```

7. Use Performance Auditing Tools

Use Chrome Lighthouse (DevTools → Lighthouse tab) or:

- Google PageSpeed Insights
- WebPageTest

FINAL RESULT

The optimized web app:

- Loads much faster.
- Reduces file sizes via minification and image compression.
- Delays loading of non-essential content with lazy loading.
- Performs better on mobile and low-speed connections.