1. **Build a responsive web application for shopping cart with registration, login, catalog and cart pages using CSS3 features, flex and grid.**

AIM:

Build a responsive web application for shopping cart with registration, login, catalog and cart pages using CSS3 features, flex and grid.

DESCRIPTION:

HTML, CSS, and JavaScript are essential components for creating a functional responsive web application. A condensed example of a shopping cart with registration, login, catalogue, and cart pages is provided.

Project Structure:

1. **index.html** - Main HTML file containing the structure of the web application.
2. **styles.css** - CSS file for styling the web pages.
3. **script.js** - JavaScript file for handling interactions and logic.
4. **images/** - Folder for storing images.

index.html:

```
<!DOCTYPE html>
<html lang="en">
<head>
 <meta charset="UTF-8">
 <meta name="viewport" content="width=device-width, initial-scale=1.0">
 <link rel="stylesheet" href="styles.css">
 <title>Shopping Cart</title>
</head>
<body>
 <header>
  <h1>Shopping Cart</h1>
  <nav>
   <ul>
    <li><a href="#catalog">Catalog</a></li>
    <li><a href="#cart">Cart</a></li>
    <li><a href="#login">Login</a></li>
    <li><a href="#register">Register</a></li>
   </ul>
  </nav>
 </header>
 <main id="content">
  <!-- Content will be loaded dynamically using JavaScript -->
 </main>
 <script src="script.js"></script>
</body>
</html>
```

styles.css:

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

```
header {
 background-color:
 #333; color: #fff;
 padding:
 10px;
 text-align:
 center;
}
nav ul {
 list-style:
 none;
 padding: 0;
 display: flex;
 justify-content: center;
}
nav li {
 margin: 0 10px;
}
main {
 padding: 20px;
}
/* Add more styles based on your
design */
```

script.js:

```
// Dummy data for the
catalog const catalog = [
 { id: 1, name: 'Product 1', price: 20 },
 { id: 2, name: 'Product 2', price: 30 },
 { id: 3, name: 'Product 3', price: 25 },
];
// Function to load the
catalog function
loadCatalog() {
 const catalogContainer =
 document.getElementById('content');
 catalogContainer.innerHTML = '<h2>Catalog</h2>';
 catalog.forEach(product => {
  const productCard =
  document.createElement('div');
  productCard.classList.add('product-card');
  productCard.innerHTML = `
   <h3>${product.name}</h3>
   <p>$${product.price}</p>
   <button onclick="addToCart(${product.id})">Add to Cart</button>
  `;
  catalogContainer.appendChild(produ
  ctCard);
 });
}
// Function to add a product to the
cart function addToCart(productId) {
 // Implement cart functionality here
 console.log(`Product ${productId} added to
  cart`);
}
// Initial load
loadCatalog();
```

2. Make the above web application responsive web application using Bootstrap framework AIM: Make the above web application responsive web application using Bootstrap framework

DESCRIPTION:

Bootstrap is a popular CSS framework that makes it easy to create responsive web applications. The previous example can be modified using Bootstrap by following these steps:

Project Structure:

1. index.html - Main HTML file containing the structure of the web application with Bootstrap.
2. script.js - JavaScript file for handling interactions and logic (no changes from the previous example).
3. styles.css - You can include additional custom styles if needed.
4. **images/ - Folder for storing**

images. index.html:

```
<!DOCTYPE html>
<html lang="en">
<head>
 <meta charset="UTF-8">
 <meta name="viewport" content="width=device-width, initial-scale=1.0">
 <!-- Bootstrap CSS -->
 <link
href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0/dist/css/bootstrap.min.css"
rel="stylesheet">
 <!-- Custom CSS -->
 <link rel="stylesheet" href="styles.css">
 <title>Shopping Cart</title>
</head>
<body>
 <header class="bg-dark text-white text-center py-3">
  <h1>Shopping Cart</h1>
  <nav>
   <ul class="nav justify-content-center">
    <li class="nav-item"><a class="nav-link" href="#catalog">Catalog</a></li>
    <li class="nav-item"><a class="nav-link" href="#cart">Cart</a></li>
    <li class="nav-item"><a class="nav-link" href="#login">Login</a></li>
    <li class="nav-item"><a class="nav-link" href="#register">Register</a></li>
   </ul>
  </nav>
 </header>
 <main class="container mt-3" id="content">
  <!-- Content will be loaded dynamically using JavaScript -->
 </main>
 <!-- Bootstrap JS (optional, for certain features) -->
 <script
src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0/dist/js/bootstrap.bundle.min.
js"></script>
 <script src="script.js"></script>
</body>
</html>
```

3. Use JavaScript for doing client – side validation of the pages implemented in experiment 1 and experiment 2

AIM:

Use JavaScript for doing client – side validation of the pages implemented in experiment 1: Build a responsive web application for shopping cart with registration, login, catalog and cart pages using CSS3 features, flex and grid and experiment 2: Make the above web application responsive web application using Bootstrap framework

DESCRIPTION:

To perform client-side validation using JavaScript, you can add scripts to validate user inputs on the registration and login pages.

The modifications for both experiments are listed below.

Experiment 1: Responsive Web Application without Bootstrap Add the following JavaScript code to **script.js**:

```
// Function to validate
registration form function
validateRegistration() {
 const username =
 document.getElementById('username').value; const
 password =
 document.getElementById('password').value; if
 (username.trim() === '' || password.trim() === '') {
 alert('Please enter both username and password.');
  return false;
 }
 // Additional validation logic can be added as
 needed return true;
}
// Function to validate login
form function
validateLogin() {
 const username =
 document.getElementById('loginUsername').value; const
 password =
 document.getElementById('loginPassword').value; if
 (username.trim() === '' || password.trim() === '') {
  alert('Please enter both username and
  password.'); return false;
 }
 // Additional validation logic can be added as
 needed return true;
}
```

Modify the HTML login and registration forms:

```
<!-- Registration Form -->
<form onsubmit="return validateRegistration()">
 <!-- ... existing form fields ... -->
 <button type="submit">Register</button>
</form>
<!-- Login Form -->
<form onsubmit="return validateLogin()">
 <!-- ... existing form fields ... -->
 <button type="submit">Login</button>
</form>
```

Experiment 2: Responsive Web Application with Bootstrap
Add the following JavaScript code to **script.js**:

```
// Function to validate
registration form function
validateRegistration() {
 const username =
 document.getElementById('username').value; const
 password =
 document.getElementById('password').value; if
 (username.trim() === '' || password.trim() === '') {
 alert('Please enter both username and password.');
  return false;
 }
 // Additional validation logic can be added as
 needed return true;
}
// Function to validate login
form function
validateLogin() {
 const username =
 document.getElementById('loginUsername').value; const
 password =
 document.getElementById('loginPassword').value; if
 (username.trim() === '' || password.trim() === '') {
  alert('Please enter both username and
  password.'); return false;
 }
 // Additional validation logic can be added as
 needed return true;
}
```

Modify the Bootstrap login and registration forms:

```
<!-- Registration Form -->
<form onsubmit="return validateRegistration()" class="needs-validation"
novalidate>
 <!-- ... existing form fields ... -->
 <button type="submit" class="btn btn-primary">Register</button>
</form>
<!-- Login Form -->
<form onsubmit="return validateLogin()" class="needs-validation" novalidate>
 <!-- ... existing form fields ... -->
 <button type="submit" class="btn btn-primary">Login</button>
</form>
```

4. Explore the features of ES6 like arrow functions, callbacks, promises, async/await. Implement an application for reading the weather information from openweathermap.org and display the information in the form of a graph on the web page.

AIM:

Explore the features of ES6 like arrow functions, callbacks, promises, async/await. Implement an application for reading the weather information from openweathermap.org and display the information in the form of a graph on the web page.

DESCRIPTION:

To implement an application for reading weather information from OpenWeatherMap.org and displaying the information in the form of a graph, we can use JavaScript with ES6 features like arrow functions, callbacks, promises, and async/await. For simplicity, we'll use the **axios** library to make HTTP requests and **Chart.js** for creating the graph.
The inclusion of these libraries is crucial for your project.

Project Structure:

1. **index.html** - Main HTML file.
2. **script.js** - JavaScript file for handling weather data and graph creation.
3. **styles.css** - CSS file for styling.
4. **node_modules/** - Folder for library dependencies.

index.html:

```
<!DOCTYPE html>
<html lang="en">
<head>
 <meta charset="UTF-8">
 <meta name="viewport" content="width=device-width, initial-scale=1.0">
 <link rel="stylesheet" href="styles.css">
 <title>Weather Graph</title>
</head>
<body>
 <div class="container">
  <h1>Weather Graph</h1>
  <canvas id="weatherGraph" width="400" height="200"></canvas>
 </div>
 <script src="https://cdn.jsdelivr.net/npm/axios/dist/axios.min.js"></script>
 <script src="https://cdn.jsdelivr.net/npm/chart.js"></script>
 <script src="script.js"></script>
</body>
</html>
```

styles.css:

```
body {
 font-family: 'Arial',
 sans-serif; margin: 0;
 padding: 0;
 background-color: #f4f4f4;
}
.container {
 max-width: 600px;
 margin: 50px auto;
 background-color:
 #fff;
 padding: 20px;
 border-radius:
 8px;
 box-shadow: 0 0 10px rgba(0, 0, 0, 0.1);
}
h1 {
 text-align: center;
}
canvas {
 display:
 block;
 margin: 20px auto;
}
```

script.js:

```
document.addEventListener('DOMContentLoad
ed', () => { const apiKey =
'YOUR_OPENWEATHERMAP_API_KEY'; const city
= 'YOUR_CITY_NAME';
 const apiUrl =
 `https://api.openweathermap.org/data/2.5/weather?q=${city}&appid=$
{apiKey}&units=metric`;
 const fetchData = async ()
   => { try {
     const response = await
     axios.get(apiUrl); const
     weatherData = response.data;
     updateGraph(weatherData.main.
     temp);
   } catch (error) {
     console.error('Error fetching weather data:', error.message);
   }
 };
 const updateGraph = (temperature) => {
   const ctx =
   document.getElementById('weatherGraph').getContext('2d');
   new Chart(ctx, {
     type:
     'bar',
     data: {
      labels:
      ['Temperature'],
      datasets: [{
        label: 'Temperature
        (°C)', data:
        [temperature],
        backgroundColor: ['#36A2EB'],
      }],
     },
     options
```

```
      : {
      scales:
      { y: {
         beginAtZero: true,
        },
      },
    },
  });
};
fetchData();});
```

1. Develop a java stand alone application that connects with the database (Oracle / mySql) and perform the CRUD operation on the database tables.

AIM: Develop a java stand alone application that connects with the database (Oracle / mySql) and perform the CRUD operation on the database tables.

DESCRIPTION:

let's create a simple Java standalone application that connects to a MySQL database and performs CRUD (Create, Read, Update, Delete) operations on a table. For this example, we'll use JDBC (Java Database Connectivity) to interact with the MySQL database.

Prerequisites:

1. Make sure you have MySQL installed, and you know the database name, username, and password.
2. Download the MySQL JDBC driver (JAR file) from MySQL Connector/J and include it in your project.

Example Java Application:

Let's assume we have a table named **employees** with columns **id**, **name**, and **salary**.

```
import java.sql.*;
public class CRUDExample {
  // JDBC URL, username, and password of MySQL server
  private static final String JDBC_URL =
  "jdbc:mysql://localhost:3306/your_database"; private static final String
  USERNAME = "your_username";
  private static final String PASSWORD = "your_password";
  public static void main(String[] args) {
    try {
      // Step 1: Establishing a connection
      Connection connection = DriverManager.getConnection(JDBC_URL, USERNAME,
PASSWORD);
      // Step 2: Creating a statement
      Statement statement = connection.createStatement();
      // Step 3: Performing CRUD operations
      createRecord(statement, "John Doe", 50000);
      readRecords(statement);
      updateRecord(statement, 1, "John Updated", 55000);
      readRecords(statement);
      deleteRecord(statement, 1);
      readRecords(statement);
      // Step 4: Closing resources
      statement.close();
      connection.close();
    } catch (SQLException e) {
      e.printStackTrace();
    }
  }
  // Create a new record in the database
  private static void createRecord(Statement statement, String name, int salary)
throws SQLException {
    String insertQuery = "INSERT INTO employees (name, salary) VALUES ('" + name + "', " +
    salary
+ ")";
    statement.executeUpdate(insertQuery);
```

```
      System.out.println("Record created successfully.");
   }
   // Read all records from the database
   private static void readRecords(Statement statement) throws
      SQLException { String selectQuery = "SELECT * FROM
      employees";
      ResultSet resultSet = statement.executeQuery(selectQuery);
      System.out.println("ID\tName\tSalary");
      while (resultSet.next()) {
         int id = resultSet.getInt("id");
         String name =
         resultSet.getString("name"); int
         salary = resultSet.getInt("salary");
         System.out.println(id + "\t" + name + "\t" + salary);
      }
      System.out.println();
   }
   // Update a record in the database
   private static void updateRecord(Statement statement, int id, String newName,
int newSalary) throws SQLException {
      String updateQuery = "UPDATE employees SET name = '" + newName + "',
salary = " + newSalary + " WHERE id = " + id;
      statement.executeUpdate(updateQuery);
      System.out.println("Record updated
      successfully.");
   }
   // Delete a record from the database
   private static void deleteRecord(Statement statement, int id) throws
      SQLException { String deleteQuery = "DELETE FROM employees
      WHERE id = " + id; statement.executeUpdate(deleteQuery);
      System.out.println("Record deleted successfully.");
   }
}
```

2. Create an xml for the bookstore. Validate the same using both DTD and XSD AIM: Create an xml for the bookstore. Validate the same using both DTD and XSD DESCRIPTION:

Let's create an XML file for a simple bookstore and validate it using both Document Type Definition (DTD) and XML Schema Definition (XSD).

Bookstore XML File (bookstore.xml):

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE bookstore SYSTEM "bookstore.dtd">
<bookstor
e>
<book>
      <title>Introduction to XML</title>
      <author>John Doe</author>
      <price>29.99</price>
   </book>
   <book>
      <title>Web Development Basics</title>
      <author>Jane Smith</author>
      <price>39.95</price>
   </book>
   <!-- Add more book entries as needed -->
</bookstore>
```

DTD File (bookstore.dtd):

```
<!ELEMENT bookstore (book+)>
<!ELEMENT book (title, author, price)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT author (#PCDATA)>
<!ELEMENT price (#PCDATA)>
```

XSD File (bookstore.xsd):

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
   <xs:element name="bookstore" type="bookstoreType"/>
   <xs:complexType name="bookstoreType">
      <xs:sequence>
         <xs:element name="book" type="bookType" minOccurs="0"
         maxOccurs="unbounded"/>
</xs:sequence>
</xs:complexType>
<xs:complexType
name="bookType">
<xs:sequence>
         <xs:element name="title" type="xs:string"/>
         <xs:element name="author" type="xs:string"/>
         <xs:element name="price" type="xs:decimal"/>
</xs:sequence>
</xs:complexType>
</xs:schema
>
```

3. Design a controller with servlet that provides the interaction with application developed in experiment 1 and the database created in experiment 5

AIM: Design a controller with servlet that provides the interaction with application developed in experiment 1: Build a responsive web application for shopping cart with registration, login, catalog and cart pages using CSS3 features, flex and grid and the database created in experiment 5: Develop a java stand alone application that connects with the database (Oracle / mySql) and perform the CRUD operation on the database tables

DESCRIPTION:

To design a servlet controller that interacts with the shopping cart application (Experiment 1) and the database (Experiment 5), you would typically handle HTTP requests from the web application, process the data, and communicate with the database to perform CRUD operations.

This is a basic servlet controller example, but in a real-world scenario, additional security measures, error handling, and Spring MVC frameworks may be needed.

Servlet Controller (ShoppingCartController.java):

```
import
java.io.IOException;
import
java.io.PrintWriter;
import
javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import
javax.servlet.http.HttpServletRequest;
import
javax.servlet.http.HttpServletResponse
;
@WebServlet("/ShoppingCartControll
er")
public class ShoppingCartController extends HttpServlet {
  protected void doPost(HttpServletRequest request, HttpServletResponse
response) throws IOException {
      String action =
      request.getParameter("action"); if
      (action != null) {
        switch
          (action) {
          case
          "register":
            handleRegistration(request,
            response); break;
          case "login":
            handleLogin(request,
            response); break;
          case "addToCart":
            handleAddToCart(request,
            response); break;
          // Add more cases for other actions as
          needed default:
            response.sendError(HttpServletResponse.SC_BAD_REQUEST, "Invalid
            action");
        }
        } else {
          response.sendError(HttpServletResponse.SC_BAD_REQUEST, "Action
          parameter missing");
        }
      }
      private void handleRegistration(HttpServletRequest request,
    HttpServletResponse response) throws IOException {
        // Extract registration data from request
        String username = request.getParameter("username");
String password = request.getParameter("password");
        // Perform registration logic (e.g., insert data into the database)
        // Send response to the client
        PrintWriter out =
        response.getWriter();
        out.println("Registration
        successful");
      }
      private void handleLogin(HttpServletRequest request, HttpServletResponse
    response) throws IOException {
        // Extract login data from request
        String username =
        request.getParameter("username"); String
        password =
        request.getParameter("password");
        // Perform login logic (e.g., check credentials against the database)
        // Send response to the client
        PrintWriter out =
        response.getWriter();
        out.println("Login successful");
      }
      private void handleAddToCart(HttpServletRequest request,
    HttpServletResponse response) throws IOException {
        // Extract cart data from request
        String productId = request.getParameter("productId");
        // Additional parameters as needed
        // Perform logic to add the product to the user's cart (e.g., update database)
        // Send response to the client
        PrintWriter out =
        response.getWriter();
        out.println("Product added to
        cart");
      }
      // Add more methods for other actions as needed
    }
```

4. Maintaining the transactional history of any user is very important. Explore the various session tracking mechanism (Cookies, HTTP Session)

AIM: Maintaining the transactional history of any user is very important. Explore the various session tracking mechanism (Cookies, HTTP Session)

DESCRIPTION:

Session tracking mechanisms are crucial for maintaining the state of a user's interactions with a web application. Two common methods for session tracking are Cookies and HTTP Sessions.

1. **Cookies: Cookies are small data pieces stored on a user's device by a web browser, used to maintain user-specific information between the client and the server.**

Example: Suppose you want to track the user's language preference.

Server-side (in a web server script, e.g., in Python with Flask):

```
from flask import Flask, request,
render_template, make_response app = Flask(_name_)
@app.route('/
') def index():
  # Check if the language cookie is set
  user_language = request.cookies.get('user_language')
  return render_template('index.html', user_language=user_language)
@app.route('/set_language/<language>')
def set_language(language):
  # Set the language preference in a cookie
  response = make_response(render_template('set_language.html'))
  response.set_cookie('user_language', language)
  return response
if _name__== '_main_':
  app.run(debug=True)
```

HTML Templates (index.html and set_language.html):

```
<!-- index.html -->
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Cookie Example</title>
</head>
<body>
  <h1>Welcome to the website!</h1>
  {% if user_language %}
    <p>Your preferred language is: {{ user_language }}</p>
  {% else %}
    <p>Your language preference is not set.</p>
  {% endif %}
  <p><a href="/set_language/en">Set language to English</a></p>
  <p><a href="/set_language/es">Set language to Spanish</a></p>
</body>
</html>
<!-- set_language.html -->
<!DOCTYPE html>
```

```
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Set Language</title>
</head>
<body>
  <h2>Language set successfully!</h2>
  <p><a href="/">Go back to the home page</a></p>
</body>
</htm
l>
```

**Outpu
t:**

When a user visits the site for the first time, the language preference is not set. When the user clicks on "Set language to English" or "Set language to Spanish," the preference is stored in a cookie.

On subsequent visits, the site recognizes the user's language preference based on the cookie.

2. HTTP Session: An HTTP session is a way to store information on the server side between requests from the same client. Each client gets a unique session ID, which is used to retrieve session data.

Example: Suppose you want to track the number of visits for each user. Server-side (in a web server script, e.g., in Python with Flask):

```
from flask import Flask, request,
render_template, session app = Flask(_name_)
app.secret_key = 'super_secret_key' # Set a secret key for session
management @app.route('/')
def index():
  # Increment the visit count in the session
  session['visit_count'] =
  session.get('visit_count', 0) + 1
  return render_template('index_session.html',
visit_count=session['visit_count']) if _name___== '_main_':
  app.run(debug=True)
```

HTML Template (index_session.html):

```
<!-- index_session.html -->
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Session Example</title>
</head>
<body>
  <h1>Welcome to the website!</h1>
  <p>This is your visit number: {{ visit_count }}</p>
</body>
</html>
```

5. Create a custom server using http module and explore the other modules of Node JS like OS, path, event

AIM: Create a custom server using http module and explore the other modules of Node JS like OS, path, event

DESCRIPTION:

Let's create a simple custom server using the **http** module in Node.js and then explore the **os**, **path**, and **events** modules with examples.

1. Creating a Custom Server with http Module:

Server-side (server.js):

```
const http = require('http');
const server = http.createServer((req, res) => {
  res.writeHead(200, {'Content-Type': 'text/plain'});
  res.end('Hello, this is a custom server!');
});
const PORT = 3000;
server.listen(PORT, () =>
{
  console.log(`Server is listening on port ${PORT}`);
});
```

To run the server: `node server.js`

Output: Visit **http://localhost:3000** in your browser or use a tool like **curl** or **Postman** to make a request, and you should see the response "Hello, this is a custom server!".

2. Exploring Node.js Modules:

A. **os** Module:

The **os** module provides operating system-related utility methods and properties.

Example:

```
const os = require('os');
console.log('OS Platform:', os.platform());
console.log('OS Architecture:', os.arch());
console.log('Total Memory (in bytes):',
os.totalmem()); console.log('Free Memory
(in bytes):', os.freemem());
```

Output: This will output information about the operating system platform, architecture, total memory, and free memory.

B. **path** Module:

The **path** module provides methods for working with file and directory paths.

Example:

```
const path = require('path');
const filePath = '/path/to/some/file.txt';
console.log('File Name:',
path.basename(filePath));
console.log('Directory Name:',
path.dirname(filePath)); console.log('File
Extension:', path.extname(filePath));
```

Output: This will output the file name, directory name, and file extension for the given file path.

C. **events** Module:

The **events** module provides an implementation of the EventEmitter pattern, allowing objects to emit and listen for events.

Example:

```
const EventEmitter =
require('events'); class MyEmitter
extends EventEmitter {} const
myEmitter = new MyEmitter();
```

```
// Event listener
myEmitter.on('customEvent',
(arg) => {
  console.log('Event triggered with argument:', arg);
});
// Emitting the event
myEmitter.emit('customEvent', 'Hello, EventEmitter!');
```

6. Develop an express web application that can interact with REST API to perform CRUD operations on student data. (Use Postman)

AIM: Develop an express web application that can interact with REST API to perform CRUD operations on student data. (Use Postman)

DESCRIPTION:

Let's create a simple Express web application that interacts with a REST API to perform CRUD (Create, Read, Update, Delete) operations on student data. We'll use MongoDB as the database and Mongoose as the ODM (Object Data Modeling) library.

Prerequisites:

1. Node.js installed on your machine.
2. MongoDB installed and running.

Steps:

1. Initialize a new Node.js project and install necessary packages.

```
mkdir
express-rest-api cd
express-rest-api
npm init -y
npm install express mongoose body-parser
```

2. Create an **app.js** file and set up Express.

```
// app.js
const express =
require('express'); const
mongoose =
require('mongoose');
const bodyParser =
require('body-parser'); const app =
express();
const PORT = 3000;
// Connect to MongoDB
mongoose.connect('mongodb://localhost:27017/studentsDB', { useNewUrlParser:
true, useUnifiedTopology: true });
mongoose.connection.on('error', console.error.bind(console, 'MongoDB connection
error:'));
// Middleware
app.use(bodyParser.json
());
// Routes
const studentRoutes =
require('./routes/studentRoutes');
app.use('/students', studentRoutes);
// Start the server
app.listen(PORT, () => {
  console.log(`Server is running on http://localhost:${PORT}`);
});
```

3. Create a **models** folder and define the **Student** model using Mongoose.

```
// models/student.js
```

```
const mongoose = require('mongoose');
const studentSchema = new
  mongoose.Schema({ name: { type: String,
  required: true },
  age: { type: Number, required:
  true }, grade: { type: String,
  required: true },
});
const Student = mongoose.model('Student',
studentSchema); module.exports = Student;
```

4.Create a routes folder and define the CRUD routes in studentRoutes.js.

```
// routes/studentRoutes.js
const express =
require('express'); const
router = express.Router();
const Student = require('../models/student');
// Create a new student
```

```
 router.post('/', async (req, res)
 => {
   try {
     const student = new
     Student(req.body); await
     student.save();
     res.status(201).send(student);
   } catch (error) {
     res.status(400).send(err
     or);
   }
 });
 // Get all students
 router.get('/', async (req,
   res) => { try {
     const students = await
     Student.find();
     res.send(students);
   } catch (error) {
     res.status(500).send(err
     or);
}
});
 // Get a student by ID
 router.get('/:id', async (req,
 res) => {
   try {
     const student = await
     Student.findById(req.params.id); if (!student)
     {
       return res.status(404).send({ error: 'Student not found' });
     }
     res.send(student);
   } catch (error) {
     res.status(500).send(err
     or);
}
});
 // Update a student by ID
 router.patch('/:id', async (req,
 res) => {
   const allowedUpdates = ['name', 'age',
   'grade']; const updates =
   Object.keys(req.body);
   const isValidOperation = updates.every(update =>
   allowedUpdates.includes(update)); if (!isValidOperation) {
     return res.status(400).send({ error: 'Invalid updates' });
   }
   try {
     const student = await Student.findByIdAndUpdate(req.params.id,
 req.body, { new: true, runValidators: true });
     if (!student) {
       return res.status(404).send({ error: 'Student not found' });
```

```
        }
        res.send(student);
    } catch (error) {
        res.status(400).send(err
        or);
}
});
  // Delete a student by ID
  router.delete('/:id', async (req,
  res) => {
    try {
        const student = await
        Student.findByIdAndDelete(req.params.id); if
        (!student) {
            return res.status(404).send({ error: 'Student not found' });
        }
        res.send(student);
    } catch (error) {
        res.status(500).send(err
        or);
    }
  });
  module.exports = router;
```

7. For the above application create authorized end points using JWT (JSON Web Token) AIM: For the above application create authorized end points using JWT (JSON Web Token) **DESCRIPTION:**

To add JWT (JSON Web Token) authentication to your Express application, you can use the jsonwebtoken package. Below are the steps to enhance the existing application with authorized endpoints using JWT:

1. Install the jsonwebtoken package:

```
npm install jsonwebtoken
```

2. Update the app.js file to include JWT authentication:

```
// app.js
const express =
require('express'); const
mongoose =
require('mongoose');
const bodyParser =
require('body-parser'); const jwt =
require('jsonwebtoken');
const app =
express(); const
PORT = 3000;
// Connect to MongoDB
mongoose.connect('mongodb://localhost:27017/studentsDB', { useNewUrlParser:
true, useUnifiedTopology: true });
mongoose.connection.on('error', console.error.bind(console, 'MongoDB connection
error:'));
// Middleware
app.use(bodyParser.json
());
// JWT Secret Key (Keep it secure, and consider using
environment variables) const JWT_SECRET = 'your_secret_key';
// JWT Authentication Middleware
const authenticateJWT = (req, res,
  next) => { const token =
  req.header('Authorization'); if
  (!token) {
    return res.status(401).json({ error: 'Unauthorized' });
  }
  try {
    const decoded = jwt.verify(token,
    JWT_SECRET); req.user = decoded;
    next();
  } catch (error) {
    res.status(403).json({ error: 'Invalid token' });
  }
};
// Routes
const authRoutes = require('./routes/authRoutes');
const studentRoutes = require('./routes/studentRoutes');
// Unprotected route for
authentication app.use('/auth',
authRoutes);
// Protected routes using JWT authentication
```

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

3. Create a new authRoutes.js file for authentication:

```
// routes/authRoutes.js
const express =
require('express'); const
router = express.Router();
const jwt =
require('jsonwebtoken');
// JWT Secret Key (Keep it secure, and consider using
environment variables) const JWT_SECRET = 'your_secret_key';
// Mock user (you might replace this with actual user
authentication logic) const mockUser = {
  username: 'admin',
  password: 'admin123',
};
// Authentication endpoint (generates a JWT token)
router.post('/login', (req, res) => {
  const { username, password } = req.body;
  if (username === mockUser.username && password ===
    mockUser.password) { const token = jwt.sign({ username },
    JWT_SECRET, { expiresIn: '1h' }); res.json({ token });
  } else {
    res.status(401).json({ error: 'Invalid credentials' });
}
});
module.exports = router;
```

4. Update the studentRoutes.js file to include authorization:

```
// routes/studentRoutes.js
const express =
require('express'); const
router = express.Router();
const Student = require('../models/student');
// ...
// Create a new student (Protected)
router.post('/', async (req, res) => {
  try {
    const student = new
    Student(req.body); await
    student.save();
    res.status(201).send(student);
  } catch (error) {
    res.status(400).send(err
    or);
}
});
// Get all students
(Protected) router.get('/',
async (req, res) => {
  try {
    const students = await
    Student.find();
    res.send(students);
});
  } catch (error) {
    res.status(500).send(err
    or);
  }
```

```
});
// ...
module.exports = router;
```

5. Test with Postman:

1. Authentication:
    - Send a POST request to http://localhost:3000/auth/login with JSON body containing the username and password. You will receive a JWT token in the response.

2. Access Protected Endpoints:

- Copy the JWT token.
    - Include the token in the Authorization header of your requests to protected endpoints (http://localhost:3000/students).
    - Ensure you're including the token with the Bearer scheme, like: Bearer your_token_here.

This implementation adds JWT authentication to your Express application. Make sure to handle real user authentication securely and consider using environment variables for sensitive information like the JWT secret key. This example provides a basic structure, and you may need to adapt it based on your specific requirements.

8. Create a react application for the student management system having registration, login, contact, about pages and implement routing to navigate through these pages.

Aim: Create a react application for the student management system having registration, login, contact, about pages and implement routing to navigate through these pages.

Description:

Below is a step-by-step guide to create a simple React application for a student management system with registration, login, contact, and about pages. We'll use react-router-dom for routing.

1.Set Up a New React App:

```
npx create-react-app
student-management-system cd
student-management-system
```

2.Install react-router-dom: `npm install react-router-dom`

3. Create Components for Each Page:

src/components/Home.js:

```
//
src/components/Home.j
s import React from
'react'; const Home = ()
=> {
 return (
  <div>
   <h1>Welcome to Student Management System</h1>
  </div>
 );
};
export default Home;
```

src/components/Register.js:

```
//
src/components/Registe
r.js import React from
'react'; const Register =
() => {
 return (
  <div>
   <h1>Registration Page</h1>
   {/* Add registration form components */}
  </div>
 );
};
export default Register;
```

src/components/Login.js:

```
//
src/components/Login.j
s import React from
'react'; const Login = ()
=> {
 return (
   <div>
     <h1>Login Page</h1>
     {/* Add login form components */}
   </div>
 );
};
export default Login;
```

src/components/Contact .js:

```
//
src/components/Contact
.js import React from
'react'; const Contact = ()
=> {
 return (
   <div>
     <h1>Contact Page</h1>
     {/* Add contact form or information */}
   </div>
 );
};
export default Contact;
```

src/components/About.j
s:

```
//
src/components/About.j
s import React from
'react'; const About = ()
=> {
 return (
   <div>
     <h1>About Page</h1>
     <p>This is the about page of the Student Management System.</p>
   </div>
 );
};
export default About;
```

4.Create a src/App.js file for App Component and Routing:

```
// src/App.js
import React from 'react';
import { BrowserRouter as Router, Route, Link } from
'react-router-dom'; import Home from
'./components/Home';
import Register from
'./components/Register'; import Login
from './components/Login'; import
Contact from './components/Contact';
import About from
'./components/About'; const App = ()
=> {
 return (
   <Router>
     <div>
       <nav>
         <ul>
           <li><Link to="/">Home</Link></li>
           <li><Link to="/register">Register</Link></li>
           <li><Link to="/login">Login</Link></li>
           <li><Link to="/contact">Contact</Link></li>
           <li><Link to="/about">About</Link></li>
         </ul>
```

```
        </nav>
        <hr />
        <Route exact path="/" component={Home} />
        <Route path="/register" component={Register} />
```

```
      <Route path="/login" component={Login} />
      <Route path="/contact" component={Contact} />
      <Route path="/about" component={About} />
     </div>
   </Router>
 );
};
export default App;
```

5.Start the React App: npm start

Visit http://localhost:3000 in your browser, and you should see the navigation links for Home, Register, Login, Contact, and About pages. Clicking on the links will navigate to the respective pages.

This is a basic structure, and you can enhance it by adding functionality to the registration and login forms, managing state, and connecting to a backend for data storage. Also, consider using react-router-dom features like Switch, Redirect, and withRouter for more advanced routing scenarios.

9. Create a service in react that fetches the weather information from openweathermap.org and the display the current and historical weather information using graphical representation using chart.js

AIM: Create a service in react that fetches the weather information from openweathermap.org and the display the current and historical weather information using graphical representation using chart.js

DESCRIPTION:

To achieve this, you can create a React service that fetches weather information from OpenWeatherMap API and uses Chart.js for graphical representation.

1. Set Up the React App:

```
npx create-react-app
weather-app cd
weather-app
```

2. Install Dependencies: `npm install axios react-chartjs-2`

3. Create a WeatherService Component:

src/services/WeatherService.js

```
//
src/services/WeatherServic
e.js import axios from
'axios';
const API_KEY = 'YOUR_OPENWEATHERMAP_API_KEY';
const WeatherService = {
 getCurrentWeather: async
 (city) => { try {
    const response = await
    axios.get(`http://api.openweathermap.org/data/2.5/weather?q=$
{city}&appid=${API_KEY
    }`); return
    response.data;
  } catch
    (error) {
    throw
    error;
  }
 },
 getHistoricalWeather: async (city, startDate,
   endDate) => { try {
    const response = await
axios.get(`http://api.openweathermap.org/data/2.5/onecall/timemachine?lat=${c
ity.lat}&lon=$
{city.lon}&start=${startDate}&end=${endDate}&appid=${API_KEY}`
    );
    return response.data;
  } catch
    (error) {
    throw
    error;
  }
 },
};
export default WeatherService;
```

4. **Create a WeatherChart Component: src/components/WeatherChart.js:**

```
// src/components/WeatherChart.js
import React, { useState, useEffect } from
'react'; import { Line } from 'react-chartjs-2';
import WeatherService from '../services/WeatherService';
```

```
const WeatherChart = () => {
 const [chartData, setChartData] =
 useState({}); useEffect(() => {
  const fetchData = async ()
   => { try {
    const city = { lat: 37.7749, lon: -122.4194 }; // Replace with the coordinates of
    your desired city const currentDate = Math.round(new Date().getTime() /
    1000);
    const historicalData = await WeatherService.getHistoricalWeather(city,
currentDate - 86400 * 7, currentDate);
    const labels = historicalData.hourly.map(entry => new Date(entry.dt *
1000).toLocaleTimeString([], { hour: '2-digit' }));
    const temperatures = historicalData.hourly.map(entry => entry.temp -
273.15); // Convert from Kelvin to Celsius
    setChartDat
     a({ labels,
     datasets: [
      {
       label: 'Temperature
       (°C)', data:
       temperatures,
       backgroundColor:
       'rgba(75,192,192,0.2)', borderColor:
       'rgba(75,192,192,1)', borderWidth: 1,
      },
     ],
    });
   } catch (error) {
    console.error('Error fetching weather data:', error);
   }
  };
  fetchData();
 }, []);
 return (
  <div>
   <h2>Historical Weather Chart</h2>
   <Line data={chartData} />
  </div>
 );
};
export default WeatherChart;
```

5. Integrate WeatherChart in App Component: src/App.js:

```
// src/App.js
import React from 'react';
import WeatherChart from
'./components/WeatherChart'; function App() {
 return (
  <div className="App">
   <WeatherChart />
   </div>
  );
}
export default App;
```

6. Add OpenWeatherMap API Key:
Replace 'YOUR_OPENWEATHERMAP_API_KEY' in src/services/WeatherService.js with your actual OpenWeatherMap API key.
7. Run the App: npm start
Visit http://localhost:3000 in your browser. The app should fetch historical weather data and display it using Chart.js.
Note: The provided example fetches historical weather data for the past 7 days. You may customize the date range and other parameters based on your requirements. Additionally, consider error handling and user interface improvements for a production-ready application**.**

10. Create a TODO application in react with necessary components and deploy it into github

AIM: Create a TODO application in react with necessary components and deploy it into github

DESCRIPTION:

Let's create a simple TODO application in React and deploy it to GitHub Pages.

1. Set Up the React App:

```
npx create-react-app
todo-app cd todo-app
```

2. Create Necessary Components:

src/components/TodoForm .js:

```
//
src/components/TodoForm.js
import React, { useState } from
'react'; const TodoForm = ({
addTodo }) => { const [text,
setText] = useState(''); const
handleSubmit = (e) => {
  e.preventDefault
  (); if (text.trim()
  !== '') {
  addTodo(text);
  setText('');
  }
 };
 return (
  <form onSubmit={handleSubmit}>
   <input type="text" value={text} onChange={(e) => setText(e.target.value)}
placeholder="Add a new todo" />
   <button type="submit">Add Todo</button>
  </form>
 );
};
export default
TodoForm;
```

src/components/TodoLi st.js:

```
//
src/components/TodoLi
st.js import React from
'react';
const TodoList = ({ todos,
 deleteTodo }) => { return (
  <ul>
   {todos.map((todo) => (
    <li key={todo.id}>
     {todo.text}
     <button onClick={() => deleteTodo(todo.id)}>Delete</button>
    </li>
   ))}
  </ul>
 );
};
export default TodoList;
```

src/App.js:

```
// src/App.js
import React, { useState } from 'react';
import TodoForm from './components/TodoForm';
import TodoList from
'./components/TodoList'; function
App() {
 const [todos, setTodos] =
 useState([]); const addTodo =
 (text) => {
  setTodos([...todos, { id: Date.now(), text }]);
 };
 const deleteTodo = (id) => {
  setTodos(todos.filter((todo) => todo.id !==
  id));
 };
 return (
  <div className="App">
   <h1>TODO App</h1>
   <TodoForm addTodo={addTodo} />
   <TodoList todos={todos} deleteTodo={deleteTodo} />
  </div>
 );
}
export default App;
```

3. Create a GitHub Repository: Create a new repository on GitHub.
4. Initialize Git and Push to GitHub:

```
git init
git
add .
git commit -m "Initial commit"
git remote add origin
<your-github-repo-url> git push -u
origin master
```

5. Install GitHub Pages: `npm install gh-pages --save-dev`
6. Add Deploy Script to package.json:
Update your package.json with the following:

```
"scripts": {
 "start": "react-scripts
 start", "build":
 "react-scripts build",
 "predeploy": "npm run
 build", "deploy": "gh-pages
 -d build", "test":
 "react-scripts test", "eject":
 "react-scripts eject"
}
```

7. Deploy to GitHub Pages: `npm run deploy`
8. Access Your Deployed App:
Visit https://<username>.github.io/<repository-name> in your browser to see your deployed TODO app.
Replace <username> with your GitHub username and <repository-name> with the name of your GitHub repository.
Now, you have a simple TODO application deployed on GitHub Pages. Users can add and delete tasks directly on the deployed app.

B. Tech. II Year II Sem.

NODE JS/ REACT JS/ DJANGO

L T P C
0 0 2 1