R22 B. Tech. CSE Syllabus **NODE JS/ REACT JS/ DJANGO**

B. Tech. II Year II Sem. **L T P C**

0 0 2 1

Week-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; }
```

Script.js:

```
// Dummy data for the catalogconst
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 catalogfunction 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>$${product.price}</p> <button onclick="addToCart(${product.id})">Add to Cart</button> `; catalogContainer.appendChild(productCard); });}// Function to add a product to the cartfunction addToCart(productId) { // Implement cart functionality here console.log(`Product ${productId} added to cart`);}// Initial loadloadCatalog();background-color: #333; color: #fff; padding: 10px; text-align: center;}nav ul { list-style: none; padding: 0; display: flex; justify-content: center;}nav li { margin: 0 10px;}main { padding: 20px;}/* Add more styles based on your design */script.js:// Dummy data for the catalogconst catalog = [ { id: 1, name: 'Product 1', price: 20 }, { id: 2, name: 'Product 2', price: 30 }, { id: 3, name: 'Product 3', price: 25 },];// Function to load the catalogfunction loadCatalog() { const catalogContainer = document.getElementById('content'); catalogContainer.innerHTML = '<h2>Catalog</h2>'; catalog.forEach(product => { const productCard = document.createElement('div'); productCard.classList.add('product-card'); productCard.innerHTML = ` <h3>${product.name}</h3> <p>$${product.price}</p> <button onclick="addToCart(${product.id})">Add to Cart</button> `; catalogContainer.appendChild(productCard); });}// Function to add a product to the cartfunction addToCart(productId) { // Implement cart functionality here console.log(`Product ${productId} added to cart`);}// Initial loadloadCatalog();
```

Explanation:

1 : HTML Structure: The HTML file consists of a header, navigation, and a main content area, initially empty but dynamically populated using JavaScript.

2. CSS Styles: The CSS file offers basic styling for header, navigation, and main content area, which can be customized according to your design needs.

3. JavaScript Logic: The JavaScript file contains dummy catalog data and functions for loading and adding products to the cart, allowing real-world interaction with a server for catalog data retrieval and user cart management.

Output:

The shopping cart application's basic structure is displayed in index.html, with the catalog section containing dummy product data.

Note: The example is intentionally simplified, and you would need to add more functionality, such as user authentication, cart management, and server communication, for a fully functional shopping cart application.

WEEK-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>
```

Styles.css:

```
/* You can include additional custom styles here if needed */
```

Explanation:

1. Bootstrap Integration: In the <head> section, we added links to the Bootstrap CSS and JS files from a CDN (Content Delivery Network). This allows us to use Bootstrap's styling and functionality.

2. Bootstrap Classes: We applied Bootstrap classes to the HTML elements. For example, we used container to create a responsive fixed-width container and various utility classes for styling the header and navigation.

3. Responsive Navigation: Bootstrap's grid system and utility classes help in creating a

Responsive navigation bar. The justify-content-center class is used to center the navigation links.

4. Responsive Main Content: The container class ensures that the main content area is responsive. Bootstrap will automatically adjust the width based on the screen size.

Output:

When you open index.html in a web browser, you'll see that the web application is now responsive.

The Bootstrap framework takes care of making the layout adapt to different screen sizes, providing a more user-friendly experience on various devices.

Remember to test the responsiveness by resizing your browser or using different devices to see how the layout adjusts.

Week-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 aresponsive 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>
```

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>

</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.add EventListener('DOMContentLoaded', () => {  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: {            begin At Zero: true,          },        },      },    });  }; fetchData();

});
```

Explanation:

1. HTML Structure: We have a simple HTML structure with a container for the graph canvas.

2. CSS Styling: Basic styling to make the application look presentable.

3. JavaScript (script.js):

- The fetch Data function uses axios to make an asynchronous HTTP request to Open Weather Map API.
- The retrieved weather data is used to update the graph using the update Graph function.
- The graph is created using the Chart.js library.

4. API Key: Replace 'YOUR_OPENWEATHERMAP_API_KEY' with your Open Weather Map API key and 'YOUR_CITY_NAME' with the desired city.

Output:

When you open index.html in a web browser, the application fetches the current weather information for the specified city from Open Weather Map API and displays the temperature on a bar graph. Please note that you need to replace 'YOUR_OPENWEATHERMAP_API_KEY' with your actual API key. Additionally, the responsiveness and appearance can be further enhanced based on your design preferences.

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

AIM: Develop a java standalone 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.");    }}
```

Explanation:

1. Connection Parameters: Replace your_database, your_username, and your_password withyour actual database name, username, and password.

2. JDBC Connection:

The getConnection method is used to establish a connection to the MySQL database.

3. CRUD Operations:

createRecord: Inserts a new record into the database.

 readRecords: Reads and prints all records from the database.

updateRecord: Updates a record in the database.

deleteRecord: Deletes a record from the database.

4. Output: The application prints the records before and after each operation, showing the CRUD operations in action.

Output:

When you run this Java application, you should see output similar to the following in the console:

Record created successfully.

ID Name Salary

1 John Doe 50000

Record updated successfully.

ID Name Salary

1 John Updated 55000

Record deleted successfully.

ID Name Salary This demonstrates a simple Java standalone application for CRUD operations on a MySQL database using JDBC. Keep in mind that for a production environment, you would want to use prepared statements to prevent SQL injection and handle exceptions more gracefully

6. 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):

!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>

Explanation:

1. XML File (bookstore.xml):

Represents a simple XML structure with a list of books in a bookstore.

2. DTD File (bookstore.dtd):

Describes the structure of the XML document using Document Type Definition. Specifies that a bookstore must contain one or more book elements. Each book must contain title, author, and price elements.

3. XSD File (bookstore.xsd):

Describes the structure of the XML document using XML Schema Definition. Defines complex types for bookstore and book. Specifies that a bookstore must contain an unbounded sequence of book elements. Each book must contain title (string), author (string), and price (decimal) elements.

Validation: You can validate the XML file using a tool or programming language that supports DTD and XSD validation. Here, I'll show you how to do it using a Java program.

Java Program for Validation:

```
import java.io.File;
import java.io.FileInputStream;
import java.io.IOException;
import java.io.InputStream;
import javax.xml.XMLConstants;
import javax.xml.transform.stream.StreamSource;
import javax.xml.validation.Schema;
import javax.xml.validation.SchemaFactory;
import javax.xml.validation.Validator;
import org.xml.sax.SAXException;
public class XMLValidator {
  public static void main(String[] args) {
    validateWithDTD("bookstore.xml", "bookstore.dtd");
    validateWithXSD("bookstore.xml", "bookstore.xsd");
  }
  private static void validateWithDTD(String xmlFile, String dtdFile) {
```

```
    try {
      InputStream xmlStream = new FileInputStream(new File(xmlFile));
      InputStream dtdStream = new FileInputStream(new File(dtdFile));
       SchemaFactory schemaFactory =
       SchemaFactory.newInstance(XMLConstants.XML_DTD_NS_URI);
      Schema schema = schemaFactory.newSchema(new StreamSource(dtdStream));
      Validator validator = schema.newValidator();
      validator.validate(new StreamSource(xmlStream));
      System.out.println("Validation with DTD successful.");
    } catch (SAXException | IOException e) {
      System.out.println("Validation with DTD failed. Reason: " + e.getMessage());
    }
  }
private static void validateWithXSD(String xmlFile, String xsdFile)
{
try {
InputStream xmlStream = new FileInputStream(new File(xmlFile));
InputStream xsdStream = new FileInputStream(new File(xsdFile));
SchemaFactory schemaFactory =
SchemaFactory.newInstance(XMLConstants.W3C_XML_SCHEMA_NS_URI);
Schema schema = schemaFactory.newSchema(new StreamSource(xsdStream));
Validator validator = schema.newValidator();
validator.validate(new StreamSource(xmlStream));
System.out.println("Validation with XSD successful.");
}
catch (SAXException | IOException e) {
System.out.println("Validation with XSD failed. Reason: " + e.getMessage());
}
}
}
```

Output:

If the XML file adheres to the specified DTD and XSD, you will see output messages like:

Validation with DTD successful.

Validation with XSD successful.

If there are issues with the XML file, the program will print error messages explaining the validation failure.

Make sure to replace "bookstore.xml", "bookstore.dtd", and "bookstore.xsd" with the actual file paths in your project. Also, you can use various online XML validators to validate your XML files against DTD and XSD if you prefer a web-based approach.

7. 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 standalone 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("/ShoppingCartController") 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
}
```

Explanation:

1. Servlet Annotation (@WebServlet("/ShoppingCartController")):

This annotation maps the servlet to the specified URL pattern.

2. doPost Method: Handles HTTP POST requests. It checks the value of the "action" parameter in the request and calls the appropriate method based on the action.

3. Action Methods (handleRegistration, handleLogin, handleAddToCart):

Each method handles a specific action requested by the client. It extracts data from the request, performs the necessary logic (e.g., database operations), and sends a response back to the client.

4. Error Handling: The controller checks for invalid actions or missing parameters and sends an error response if necessary.

Note: This example assumes that the web application sends HTTP POST requests to the servlet with an "action" parameter to specify the desired operation.

Ensure that your web application sends requests to the correct URL pattern, in this case, "/ShoppingCartController".Make sure to handle exceptions properly, and consider using a connection pool for database connections in a production environment.

This is a basic outline, and depending on your specific application requirements, you may need to expand and customize these methods accordingly.

8. 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>
</html>
```

Output:

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):

```
<!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>
```

Output:

- Each time a user visits the site, the server increments the visit count stored in the session.
- The visit count is unique to each user and persists across multiple requests until the session expires.

These examples demonstrate simple use cases for cookies and HTTP sessions to maintain user-specific information on the client and server sides, respectively.

9. 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 Event Emitter pattern, allowing objects to emit and listen for events.

Example:

const EventEmitter = require('events');

class MyEmitter extends EventEmitter {}

const myEmitter = new MyEmitter();