SKILL DEVELOPMENT LAB

List of Experiments

Course Code: KG21CS518 **B. Tech. III Year I - Semester**

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

Source Code :

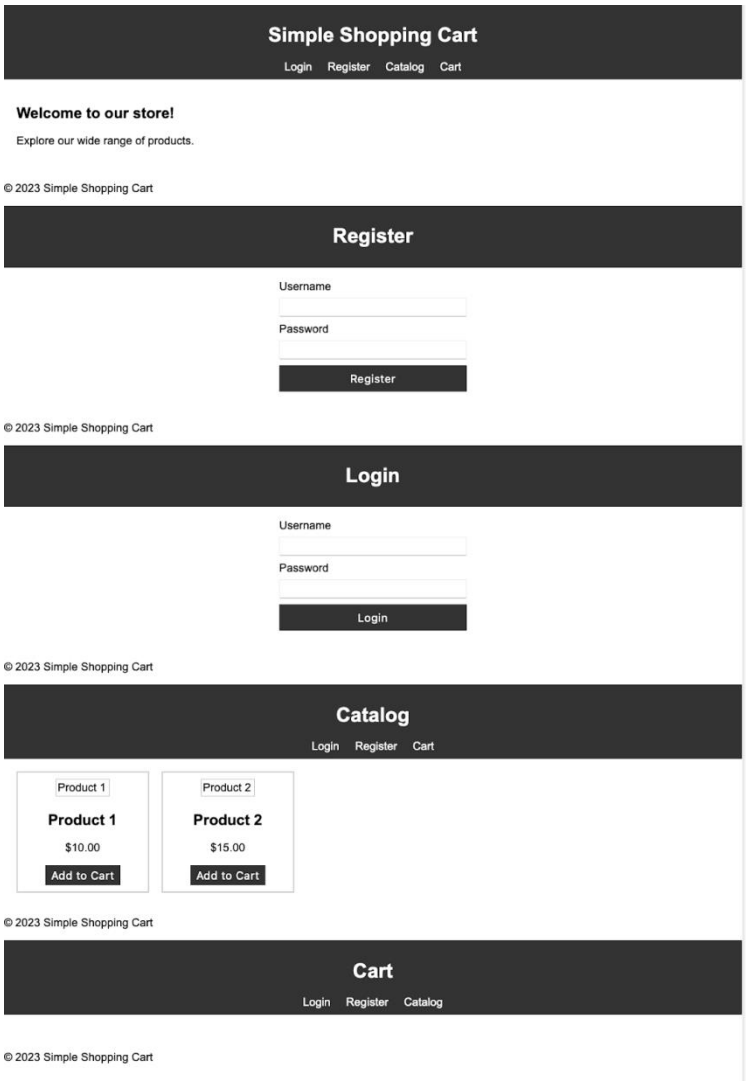
```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Simple Shopping Cart</title>
    <link rel="stylesheet" href="styles.css">
</head>
<body>
    <header>
        <h1>Simple Shopping Cart</h1>
        <nav>
            <a href="login.html">Login</a>
            <a href="register.html">Register</a>
            <a href="catalog.html">Catalog</a>
            <a href="cart.html">Cart</a>
        </nav>
    </header>

    <main>
        <h2>Welcome to our store!</h2>
        <p>Explore our wide range of products.</p>
    </main>

    <footer>
        <p>&copy; 2023 Simple Shopping Cart</p>
    </footer>
</body>
</html>
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
```

```
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Register - Simple Shopping Cart</title>
    <link rel="stylesheet" href="styles.css">
</head>
<body>
    <header>
        <h1>Register</h1>
    </header>

    <main>
        <form action="#">
            <label for="username">Username</label>
            <input type="text" id="username" name="username" required>
            <label for="password">Password</label>
            <input type="password" id="password" name="password" required>
            <input type="submit" value="Register">
        </form>
    </main>

    <footer>
        <p>&copy; 2023 Simple Shopping Cart</p>
    </footer>
</body>
</html>
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Login - Simple Shopping Cart</title>
    <link rel="stylesheet" href="styles.css">
</head>
<body>
    <header>
        <h1>Login</h1>
    </header>

    <main>
        <form action="#">
            <label for="username">Username</label>
            <input type="text" id="username" name="username" required>
            <label for="password">Password</label>
            <input type="password" id="password" name="password" required>
            <input type="submit" value="Login">
        </form>
    </main>

    <footer>
```

```
        <p>&copy; 2023 Simple Shopping Cart</p>
    </footer>
</body>
</html>
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Catalog - Simple Shopping Cart</title>
    <link rel="stylesheet" href="styles.css">
</head>
<body>
    <header>
        <h1>Catalog</h1>
        <nav>
            <a href="login.html">Login</a>
            <a href="register.html">Register</a>
            <a href="cart.html">Cart</a>
        </nav>
    </header>

    <main class="product-list">
        <div class="product">
            <img src="product1.jpg" alt="Product 1">
            <h2>Product 1</h2>
            <p>$10.00</p>
            <button>Add to Cart</button>
        </div>

        <div class="product">
            <img src="product2.jpg" alt="Product 2">
            <h2>Product 2</h2>
            <p>$15.00</p>
            <button>Add to Cart</button>
        </div>

        <!-- Add more products as needed -->
    </main>

    <footer>
        <p>&copy; 2023 Simple Shopping Cart</p>
    </footer>
</body>
</html>
<!DOCTYPE html>
<html lang="en">
<head>
```

```
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Cart - Simple Shopping Cart</title>
    <link rel="stylesheet" href="styles.css">
</head>
<body>
    <header>
        <h1>Cart</h1>
        <nav>
            <a href="login.html">Login</a>
            <a href="register.html">Register</a>
            <a href="catalog.html">Catalog</a>
        </nav>
    </header>

    <main class="cart-items">
        <!-- Cart items will be dynamically populated here using JavaScript -->
    </main>

    <footer>
        <p>&copy; 2023 Simple Shopping Cart</p>
    </footer>
</body>
</html>
```

Output:



2. **Make the above web application responsive web application using Bootstrap framework.**

Source Code:

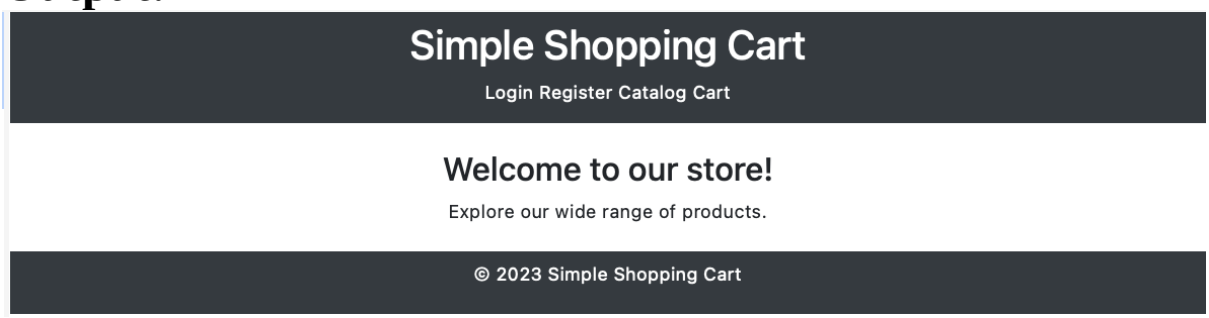
```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1, shrink-to-fit=no">
    <title>Simple Shopping Cart</title>
```

```
    <link rel="stylesheet"
href="https://stackpath.bootstrapcdn.com/bootstrap/4.5.2/css/bootstrap.min.css">
</head>
<body>
    <header class="bg-dark text-white text-center py-2">
        <h1>Simple Shopping Cart</h1>
        <nav class="mb-2">
            <a href="login.html" class="text-white">Login</a>
            <a href="register.html" class="text-white">Register</a>
            <a href="catalog.html" class="text-white">Catalog</a>
            <a href="cart.html" class="text-white">Cart</a>
        </nav>
    </header>

    <main class="container mt-4">
        <h2 class="text-center">Welcome to our store!</h2>
        <p class="text-center">Explore our wide range of products.</p>
    </main>

    <footer class="bg-dark text-white text-center py-2 mt-4">
        <p>&copy; 2023 Simple Shopping Cart</p>
    </footer>

    <script src="https://code.jquery.com/jquery-3.5.1.slim.min.js"></script>
    <script
src="https://cdn.jsdelivr.net/npm/@popperjs/core@2.9.3/dist/umd/popper.min.js"></script>
    <script
src="https://stackpath.bootstrapcdn.com/bootstrap/4.5.2/js/bootstrap.min.js"></script>
</body>
</html>
```

Script.js

```
<link rel="stylesheet"
href="https://stackpath.bootstrapcdn.com/bootstrap/4.5.2/css/bootstrap.min.css">
<script src="https://code.jquery.com/jquery-3.5.1.slim.min.js"></script>
<script
src="https://cdn.jsdelivr.net/npm/@popperjs/core@2.9.3/dist/umd/popper.min.js"></script>
<script src="https://stackpath.bootstrapcdn.com/bootstrap/4.5.2/js/bootstrap.min.js"></script>
```

Output:



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

Source Code:

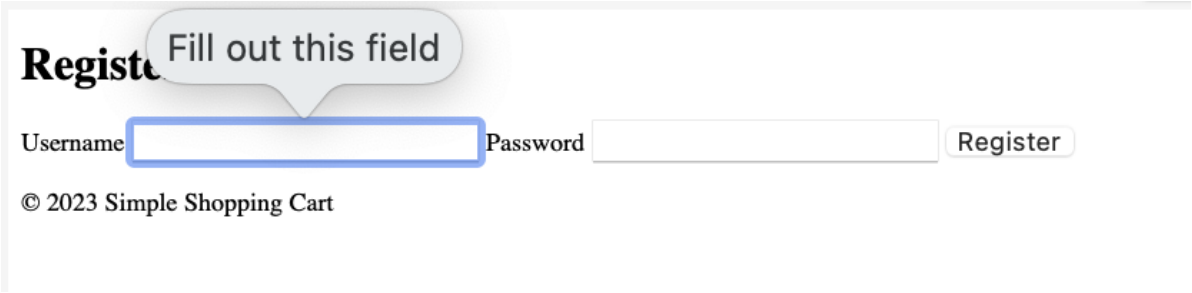
Exp1:

```
<script>
    function validateForm() {
        var username = document.getElementById("username").value;
        var password = document.getElementById("password").value;

        if (username === "" || password === "") {
            alert("Please fill in all fields.");
            return false;
        }

        return true;
    }
</script>
```

Exp2:

```
<script>
    function validateForm() {
        var username = document.getElementById("username").value;
        var password = document.getElementById("password").value;

        if (username === "" || password === "") {
            alert("Please fill in all fields.");
            return false;
        }

        return true;
    }
</script>
```

Output:



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.

To implement an application for reading weather information from openweathermap.org and displaying it as a graph on a web page, we'll utilize modern JavaScript features like arrow functions, callbacks, promises, and async/await along with a library like Chart.js for creating graphs. Below is an example implementation using Node.js for the backend and Express.js for serving the web page.

First, you need to set up a Node.js project and install necessary dependencies. Ensure you have Node.js and npm installed on your system. Then, create a new directory for your project and run:

Bash:

```
mkdir weather-graph

cd weather-graph

npm init -y

npm install express axios chart.js
```

Now, you can create your main JavaScript file (app.js) with the following content:

Javascript:

```
const express = require('express');
const axios = require('axios');
const Chart = require('chart.js');
const app = express();

const API_KEY = 'YOUR_OPENWEATHERMAP_API_KEY';

// Function to fetch weather data
const fetchWeatherData = async (city) => {
    try {
        const response = await
axios.get(`http://api.openweathermap.org/data/2.5/weather?q=${cit
y}&appid=${API_KEY}`);
        return response.data.main.temp; // Return temperature
    } catch (error) {
        console.error('Error fetching weather data:', error);
        throw error;
    }
```

```
};

// Define routes
app.get('/', async (req, res) => {
    const city = req.query.city || 'London'; // Default city is London
    try {
        const temperature = await fetchWeatherData(city);
        res.send(`
            <h1>Weather Information for ${city}</h1>
            <canvas id="weatherChart"></canvas>
            <script>
                const ctx =
document.getElementById('weatherChart').getContext('2d');
                const weatherChart = new Chart(ctx, {
                    type: 'bar',
                    data: {
                        labels: ['Temperature'],
                        datasets: [{
                            label: 'Temperature',
                            data: [${temperature}],
                            backgroundColor: 'rgba(75, 192, 192, 0.2)',
                            borderColor: 'rgba(75, 192, 192, 1)',
```

```
                    borderWidth: 1
                }]
            },
            options: {
                scales: {
                    y: {
                        beginAtZero: true
                    }
                }
            }
        });
    </script>
    `);
  } catch (error) {
    res.status(500).send('Error fetching weather data');
  }
});

// Start server
const PORT = process.env.PORT || 3000;
app.listen(PORT, () => {
```

```
    console.log(`Server is running on port ${PORT}`);
});
```

Make sure to replace 'YOUR_OPENWEATHERMAP_API_KEY' with your actual OpenWeatherMap API key.

To run the application, execute:

Bash:

```
node app.js
```

Output:

Now, if you open your browser and go to `http://localhost:3000`, you should see the weather information displayed as a graph. You can also specify a city by passing it as a query parameter, like `http://localhost:3000/?city=Paris`.

This application fetches the current temperature for a given city from OpenWeatherMap API and displays it as a bar graph using Chart.js.

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

Source Code:

Below is an example of a Java standalone application that connects to a MySQL database and performs CRUD (Create, Read, Update,

Delete) operations on a table. You need to have the MySQL JDBC driver added to your project for this to work.

```java
import java.sql.*;

public class CRUDExample {

    static final String JDBC_DRIVER = "com.mysql.jdbc.Driver";
    static final String DB_URL = "jdbc:mysql://localhost/your_database_name";

    static final String USER = "your_username";
    static final String PASS = "your_password";

    public static void main(String[] args) {
        Connection conn = null;
        Statement stmt = null;

        try {
            Class.forName(JDBC_DRIVER);
```

```
        System.out.println("Connecting to database...");
        conn = DriverManager.getConnection(DB_URL, USER,
PASS);

        System.out.println("Creating statement...");
        stmt = conn.createStatement();

        // Create
        String sql = "INSERT INTO your_table_name (id, name)
VALUES (1, 'John')";
        stmt.executeUpdate(sql);
        System.out.println("Record inserted successfully.");

        // Read
        sql = "SELECT id, name FROM your_table_name";
        ResultSet rs = stmt.executeQuery(sql);
        while (rs.next()) {
            int id = rs.getInt("id");
            String name = rs.getString("name");
            System.out.println("ID: " + id + ", Name: " + name);
```

```
            }
            rs.close();

            // Update
            sql = "UPDATE your_table_name SET name = 'Doe'
WHERE id = 1";
            int rowsUpdated = stmt.executeUpdate(sql);
            if (rowsUpdated > 0) {
                System.out.println("Update successful.");
            }

            // Delete
            sql = "DELETE FROM your_table_name WHERE id = 1";
            int rowsDeleted = stmt.executeUpdate(sql);
            if (rowsDeleted > 0) {
                System.out.println("Deletion successful.");
            }

        } catch (SQLException se) {
            se.printStackTrace();
```

```
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            try {
                if (stmt != null) stmt.close();
            } catch (SQLException se2) {
            }
            try {
                if (conn != null) conn.close();
            } catch (SQLException se) {
                se.printStackTrace();
            }
        }
    }
}
```

Output:

Make sure to replace `"your_database_name"`, `"your_username"`, `"your_password"`, and `"your_table_name"` with your actual database credentials and table name. This code includes the basic CRUD operations: Create, Read, Update, and Delete. The output will be printed to the console, showing the results of each operation.

6. Create an xml for the bookstore. Validate the same using both DTD and XSD.

Source Code:

Here's an example of an XML for a simple Bookstore, along with corresponding DTD and XSD schemas for validation:

XML (bookstore.xml):

```xml
<?xml version="1.0" encoding="UTF-8"?>
<bookstore>
  <book category="fiction">
    <title>The Great Gatsby</title>
    <author>F. Scott Fitzgerald</author>
    <year>1925</year>
    <price>12.99</price>
  </book>
  <book category="fiction">
```

```
        <title>To Kill a Mockingbird</title>
        <author>Harper Lee</author>
        <year>1960</year>
        <price>10.50</price>
    </book>
    <book category="nonfiction">
        <title>Steve Jobs</title>
        <author>Walter Isaacson</author>
        <year>2011</year>
        <price>15.00</price>
    </book>
</bookstore>
```

DTD (bookstore.dtd):

```xml
<!ELEMENT bookstore (book+)>
<!ELEMENT book (title, author, year, price)>
<!ATTLIST book category CDATA #REQUIRED>
<!ELEMENT title (#PCDATA)>
<!ELEMENT author (#PCDATA)>
<!ELEMENT year (#PCDATA)>
<!ELEMENT price (#PCDATA)>
```

XSD (bookstore.xsd):

```xml
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="bookstore">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="book" maxOccurs="unbounded">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="title" type="xs:string"/>
              <xs:element name="author" type="xs:string"/>
              <xs:element name="year" type="xs:gYear"/>
              <xs:element name="price" type="xs:decimal"/>
            </xs:sequence>
            <xs:attribute name="category" type="xs:string" use="required"/>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

You can use any XML validator to validate the XML against the DTD and XSD. Here's how you can do it using a Python script:

```python
from lxml import etree

def validate_with_dtd(xml_file, dtd_file):
    dtd = etree.DTD(open(dtd_file))
    xml = etree.parse(xml_file)
    return dtd.validate(xml)

def validate_with_xsd(xml_file, xsd_file):
    xmlschema_doc = etree.parse(xsd_file)
    xmlschema = etree.XMLSchema(xmlschema_doc)
    xml = etree.parse(xml_file)
    return xmlschema.validate(xml)

xml_file = 'bookstore.xml'
dtd_file = 'bookstore.dtd'
xsd_file = 'bookstore.xsd'

print("Validation with DTD:", validate_with_dtd(xml_file, dtd_file))
print("Validation with XSD:", validate_with_xsd(xml_file, xsd_file))
```

Output:

Make sure to have the `lxml` library installed (`pip install lxml`) to run this script. Replace `bookstore.xml`, `bookstore.dtd`, and `bookstore.xsd` with the paths to your files.

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

Source Code:

To design a servlet controller that interacts with an application and a database, we'll follow the MVC (Model-View-Controller) design pattern. In this setup, the servlet acts as the controller, mediating between the user interface (view) and the database (model).

Assuming you have an application developed in Experiment 1 and a database created in Experiment 5, we'll create a servlet controller that handles requests from the application's UI, performs necessary operations on the database, and sends back appropriate responses.

Here's a basic structure of how you can design the servlet:

```java
import java.io.*;
import javax.servlet.*;
```

```
import javax.servlet.http.*;
import java.sql.*;

public class ApplicationController extends HttpServlet {

    // JDBC URL, username, and password of MySQL server
    private static final String JDBC_URL =
"jdbc:mysql://localhost:3306/your_database";
    private static final String JDBC_USER = "your_username";
    private static final String JDBC_PASSWORD = "your_password";

    // JDBC variables for opening, closing and managing connection
    private Connection connection;

    public void init() throws ServletException {
        try {
            // Load the MySQL driver
            Class.forName("com.mysql.jdbc.Driver");
            // Establish connection to MySQL database
            connection = DriverManager.getConnection(JDBC_URL,
JDBC_USER, JDBC_PASSWORD);
        } catch (ClassNotFoundException | SQLException e) {
```

```
            e.printStackTrace();
        }
    }

    public void doGet(HttpServletRequest request,
HttpServletResponse response) throws ServletException,
IOException {
        // Handle GET requests
        // You can implement different actions based on GET requests
from the application UI
        // For example, retrieving data from the database and sending it
to the UI

        // Example:
        // String data = fetchDataFromDatabase();
        // response.getWriter().write(data);
    }

    public void doPost(HttpServletRequest request,
HttpServletResponse response) throws ServletException,
IOException {
        // Handle POST requests
        // You can implement different actions based on POST requests
from the application UI
```

```
        // For example, inserting data into the database

        // Example:
        // String newData = request.getParameter("data");
        // insertDataIntoDatabase(newData);
    }

    public void destroy() {
        try {
            // Close connection to MySQL database when servlet is
destroyed
            if (connection != null) {
                connection.close();
            }
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

In this servlet:

1. The `init()` method is used to initialize the servlet, including establishing a connection to the database.

2. The `doGet()` method handles GET requests from the application's UI. You can implement different actions here, like fetching data from the database.

3. The `doPost()` method handles POST requests from the application's UI. You can implement different actions here, like inserting data into the database.

4. The `destroy()` method is called when the servlet is destroyed, where you should close the database connection.

You'll need to replace placeholders like `your_database`, `your_username`, and `your_password` with your actual database credentials. Additionally, you'll need to implement methods for interacting with the database according to your application's requirements inside the `doGet()` and `doPost()` methods.

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

Source Code:

Certainly! Maintaining transactional history and session tracking are crucial aspects of web applications for various purposes such as

personalization, security, and analytics. Let's explore two common mechanisms for session tracking: Cookies and HTTP Session.

1. **Cookies:**

- **Definition:** Cookies are small pieces of data stored on the client's browser. They are sent with each HTTP request, allowing servers to identify users and track their sessions.

- **Usage:** When a user visits a website, the server can send a cookie to the browser, which is then stored locally. Subsequent requests from the same browser include the cookie, allowing the server to recognize the user.

- **Types:**

  - **Session Cookies:** These cookies are temporary and expire when the user closes the browser. They are often used for session management.

  - **Persistent Cookies:** These cookies are stored on the user's device for a specified duration, even after the browser is closed. They are used for long-term tracking and personalization.

- **Security Concerns:** Cookies are susceptible to security threats such as cross-site scripting (XSS) and cross-site request forgery (CSRF). Therefore, sensitive information should not be stored in cookies, and proper security measures must be implemented.

2. **HTTP Session:**

- **Definition:** HTTP session is a server-side mechanism for tracking user sessions. It maintains state information between HTTP requests from the same user.

- **Usage:** When a user interacts with a web application, the server creates a unique session identifier (session ID) and associates it with the user's session data.

- **Storage:** Session data is typically stored on the server-side, either in memory, a database, or a distributed cache.

- **Session Tracking:** The session ID is usually passed between the client and server using cookies, URL rewriting, or hidden form fields. Once the session ID is received, the server retrieves the corresponding session data.

- **Security:** HTTP sessions are generally considered more secure than cookies since session data is stored on the server-side. However, session hijacking and fixation are potential security risks that need to be mitigated through proper session management techniques.

- **Expiration:** Sessions can be configured to expire after a certain period of inactivity or at the end of a browsing session.

Both cookies and HTTP sessions have their advantages and disadvantages, and the choice between them depends on factors such as security requirements, scalability, and implementation preferences. Many modern web applications use a combination of both mechanisms to achieve the desired functionality and security.

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

Source Code:

Sure, here's an example of creating a custom server using the `http` module in Node.js and exploring other built-in modules like `os`, `path`, and `events`:

```javascript
// Importing required modules
const http = require('http');
const os = require('os');
const path = require('path');
const { EventEmitter } = require('events');

// Create an instance of EventEmitter
const eventEmitter = new EventEmitter();

// Create a custom event handler
eventEmitter.on('customEvent', (message) => {
  console.log('Custom Event Occurred:', message);
});

// Create a custom server using http module
```

```
const server = http.createServer((req, res) => {
  res.writeHead(200, { 'Content-Type': 'text/plain' });
  res.end('Hello, this is a custom server!\n');
});

// Define server port
const PORT = process.env.PORT || 3000;

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

// Explore OS module
console.log('OS Platform:', os.platform());
console.log('OS CPU Architecture:', os.arch());
console.log('OS User Info:', os.userInfo());

// Explore Path module
console.log('Absolute Path:', path.resolve(__dirname));
console.log('File Extension:', path.extname('example.txt'));
```

```
// Emit a custom event

eventEmitter.emit('customEvent', 'Hello from custom event!');
```

In this example:

- We create a custom server using the `http` module that listens on port 3000.

- We explore the `os` module to retrieve information about the operating system, such as platform, CPU architecture, and user info.

- We use the `path` module to work with file paths, resolving absolute paths, and extracting file extensions.

- We create an instance of `EventEmitter` to demonstrate event handling and emit a custom event.

- Finally, we emit a custom event with a message and observe its handling.

You can run this code in a Node.js environment to see the output and explore the functionality provided by these built-in modules.

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

Source Code:

To create an Express web application that interacts with a REST API for CRUD operations on student data, you'll first need to set up the Express framework and then define routes to handle CRUD operations. You can use

Postman to test the API endpoints. Below is an example of how you can achieve this:

1. **Setting Up Express and Dependencies:**

First, make sure you have Node.js and npm (Node Package Manager) installed on your system. Then, create a new directory for your project and initialize it as a Node.js project by running:

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

Next, install Express and other required dependencies:

```
npm install express body-parser
```

2. **Creating the Express App and Setting Up Routes:**

Create a file named `server.js` and add the following code:

```javascript
```

```
const express = require('express');
const bodyParser = require('body-parser');

const app = express();
const PORT = process.env.PORT || 3000;

// Middleware
app.use(bodyParser.json());

// Mock student data (for demonstration)
let students = [
  { id: 1, name: 'John Doe', age: 20 },
  { id: 2, name: 'Jane Smith', age: 22 }
];

// Routes
app.get('/students', (req, res) => {
  res.json(students);
});

app.post('/students', (req, res) => {
  const newStudent = req.body;
```

```
  students.push(newStudent);
  res.status(201).json(newStudent);
});

app.get('/students/:id', (req, res) => {
  const id = parseInt(req.params.id);
  const student = students.find(s => s.id === id);
  if (!student) {
    res.status(404).send('Student not found');
    return;
  }
  res.json(student);
});

app.put('/students/:id', (req, res) => {
  const id = parseInt(req.params.id);
  const updatedStudent = req.body;
  let found = false;
  students = students.map(s => {
    if (s.id === id) {
      found = true;
      return { ...s, ...updatedStudent };
```

```
    }
    return s;
  });
  if (!found) {
    res.status(404).send('Student not found');
    return;
  }
  res.json(updatedStudent);
});

app.delete('/students/:id', (req, res) => {
  const id = parseInt(req.params.id);
  const deletedStudent = students.find(s => s.id === id);
  students = students.filter(s => s.id !== id);
  if (!deletedStudent) {
    res.status(404).send('Student not found');
    return;
  }
  res.json(deletedStudent);
});

// Start the server
```

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

3. **Testing with Postman:**

- Start your Express server by running `node server.js`.
- Open Postman and test the following endpoints:
  - GET: `http://localhost:3000/students` - Retrieves all students.
  - POST: `http://localhost:3000/students` - Creates a new student (send JSON data in the body).
  - GET: `http://localhost:3000/students/:id` - Retrieves a specific student by ID.
  - PUT: `http://localhost:3000/students/:id` - Updates a specific student by ID (send JSON data in the body).
  - DELETE: `http://localhost:3000/students/:id` - Deletes a specific student by ID.

That's it! You've created an Express web application with CRUD operations on student data and tested it using Postman.

11. For the above application create authorized end points using JWT (JSON Web Token).

Source Code:

To add JWT (JSON Web Token) authentication to the Express web application for authorized endpoints, you need to implement middleware to verify the JWT token before granting access to protected routes. Below is an updated version of the previous code with JWT authentication:

First, install the necessary dependencies:

```bash
npm install jsonwebtoken
```

Now, let's integrate JWT authentication into the Express application:

```javascript
const express = require('express');
const bodyParser = require('body-parser');
const jwt = require('jsonwebtoken');

const app = express();
const PORT = process.env.PORT || 3000;
const secretKey = 'your_secret_key'; // Change this to a more secure secret key
```

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

// Mock user data (for demonstration)
const users = [
  { id: 1, username: 'admin', password: 'admin' }
];

// Middleware to verify JWT token
const verifyToken = (req, res, next) => {
  const token = req.headers['authorization'];
  if (!token) {
    return res.status(403).send('Token is not provided');
  }

  jwt.verify(token, secretKey, (err, decoded) => {
    if (err) {
      return res.status(401).send('Unauthorized');
    }
    req.user = decoded;
    next();
  });
};
```

```
// Login route to generate JWT token
app.post('/login', (req, res) => {
  const { username, password } = req.body;
  const user = users.find(u => u.username === username && u.password === password);
  if (!user) {
    return res.status(401).send('Invalid username or password');
  }

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

// Protected route, requires authentication
app.get('/students', verifyToken, (req, res) => {
  // Your logic to retrieve students data here
  res.json(students);
});

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

```

In this updated version:

- We've added a `/login` endpoint to authenticate users and generate a JWT token.
- The `verifyToken` middleware checks if the JWT token is provided in the `Authorization` header and verifies its authenticity using the secret key.
- The `/students` route is now protected and requires a valid JWT token to access. The `verifyToken` middleware is used to authenticate requests to this route.

Now, to test the protected endpoints with Postman:

1. Send a POST request to `http://localhost:3000/login` with JSON body containing username and password to obtain a JWT token.
2. Copy the token from the response.
3. Set the token as a Bearer token in the Authorization header of subsequent requests to protected endpoints, such as `http://localhost:3000/students`.

This ensures that only users with a valid JWT token can access the protected endpoints.
```

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

Source Code:

Below is a basic example of how you can set up a React application for a student management system with registration, login, contact, and about pages using React Router for navigation:

First, make sure you have Node.js and npm installed on your machine. Then you can create a new React application using Create React App:

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

Next, install React Router:

```bash
npm install react-router-dom
```

Then, replace the contents of `src/App.js` with the following code:

```javascript
```

```
import React from 'react';
import { BrowserRouter as Router, Route, Switch, Link } from 'react-router-dom';
import Registration from './components/Registration';
import Login from './components/Login';
import Contact from './components/Contact';
import About from './components/About';

function App() {
  return (
    <Router>
      <div>
        <nav>
          <ul>
            <li>
              <Link to="/">Home</Link>
            </li>
            <li>
              <Link to="/registration">Registration</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>

<Switch>
  <Route path="/registration">
    <Registration />
  </Route>
  <Route path="/login">
    <Login />
  </Route>
  <Route path="/contact">
    <Contact />
  </Route>
  <Route path="/about">
    <About />
  </Route>
  <Route path="/">
    <Home />
```

```
        </Route>
      </Switch>
    </div>
  </Router>
 );
}

function Home() {
 return <h2>Home</h2>;
}

export default App;
```

Next, create components for Registration, Login, Contact, and About pages. For example:

```javascript
// src/components/Registration.js
import React from 'react';

function Registration() {
 return <h2>Registration</h2>;
}
```

```
export default Registration;

// src/components/Login.js
import React from 'react';

function Login() {
  return <h2>Login</h2>;
}

export default Login;

// src/components/Contact.js
import React from 'react';

function Contact() {
  return <h2>Contact</h2>;
}

export default Contact;

// src/components/About.js
import React from 'react';
```

```
function About() {
  return <h2>About</h2>;
}

export default About;
```

Now, you have a basic React application set up with routing for the student management system. You can customize each component with the necessary functionality and styling as per your requirements.

13. 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

Source Code:

First, make sure you have installed Chart.js and Axios in your project:

bash

```
npm install chart.js axios
```

Then, you can create a component like this:

jsx

```
import React, { useState, useEffect } from 'react';
import Chart from 'chart.js/auto';
import axios from 'axios';

const WeatherChart = () => {
  const [weatherData, setWeatherData] = useState(null);

  useEffect(() => {
    const fetchData = async () => {
      try {
        // Fetch current weather data
```

```
        const currentWeatherResponse = await axios.get(

'https://api.openweathermap.org/data/2.5/weather?q=London&appid=YOUR_API_KEY'

        );


        // Fetch historical weather data

        const historicalWeatherResponse = await axios.get(

'https://api.openweathermap.org/data/2.5/onecall/timemachine?lat=51.51&lon=-0.13&dt=1613900400&appid=YOUR_API_KEY'

        );


        // Combine current and historical data

        setWeatherData({

          current: currentWeatherResponse.data,

          historical: historicalWeatherResponse.data,

        });

      } catch (error) {

        console.error('Error fetching weather data:', error);

      }

    };


    fetchData();

  }, []);
```

```
useEffect(() => {
  if (weatherData) {
    // Create chart
    const ctx = document.getElementById('weatherChart').getContext('2d');
    new Chart(ctx, {
      type: 'bar',
      data: {
        labels: ['Current', 'Historical'],
        datasets: [
          {
            label: 'Temperature (°C)',
            data: [
              weatherData.current.main.temp,
              weatherData.historical.current.temp,
            ],
            backgroundColor: ['#36a2eb', '#ff6384'],
          },
        ],
      },
      options: {
        scales: {
          y: {
```

```
            beginAtZero: true,
          },
        },
      },
    });
  }
 }, [weatherData]);

 return (
  <div>
   <h2>Weather Information</h2>
   <canvas id="weatherChart" width="400" height="400"></canvas>
  </div>
 );
};

export default WeatherChart;
```

Replace 'YOUR_API_KEY' with your actual OpenWeatherMap API key.

This component fetches current and historical weather data for London, combines them, and displays the temperature using Chart.js in a bar chart. You can customize it further according to your requirements.

14. Create a TODO application in react with necessary components and deploy it into GitHub.

Source Code:

Sure! Below is an example of a simple TODO application in React. We'll create necessary components such as `TodoList`, `TodoItem`, and `AddTodo`, and then deploy it to GitHub Pages.

First, create a new React application:

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

Next, replace the contents of `src/App.js` with the following code:

```jsx
import React, { useState } from 'react';
import './App.css';
import TodoList from './components/TodoList';
import AddTodo from './components/AddTodo';

function App() {
```

```
  const [todos, setTodos] = useState([]);

  const addTodo = (text) => {
    setTodos([...todos, { text }]);
  };

  const removeTodo = (index) => {
    const newTodos = [...todos];
    newTodos.splice(index, 1);
    setTodos(newTodos);
  };

  return (
    <div className="App">
      <h1>TODO App</h1>
      <AddTodo addTodo={addTodo} />
      <TodoList todos={todos} removeTodo={removeTodo} />
    </div>
  );
}

export default App;
```

```

Now, let's create the `TodoList`, `TodoItem`, and `AddTodo` components.

Create a new folder named `components` inside the `src` folder, and inside this folder, create three files: `TodoList.js`, `TodoItem.js`, and `AddTodo.js`.

Here's the code for each of these components:

`TodoList.js`:

```jsx
import React from 'react';
import TodoItem from './TodoItem';

const TodoList = ({ todos, removeTodo }) => {
  return (
    <ul>
      {todos.map((todo, index) => (
        <TodoItem key={index} index={index} todo={todo}
removeTodo={removeTodo} />
      ))}
    </ul>
```
```

```
  );
};

export default TodoList;
```

`TodoItem.js`:

```jsx
import React from 'react';

const TodoItem = ({ todo, index, removeTodo }) => {
  return (
    <li>
      {todo.text} <button onClick={() =>
removeTodo(index)}>Remove</button>
    </li>
  );
};

export default TodoItem;
```

`AddTodo.js`:

```jsx
import React, { useState } from 'react';

const AddTodo = ({ addTodo }) => {
  const [text, setText] = useState('');

  const handleSubmit = (e) => {
    e.preventDefault();
    if (!text.trim()) return;
    addTodo(text);
    setText('');
  };

  return (
    <form onSubmit={handleSubmit}>
      <input
        type="text"
        placeholder="Add todo..."
        value={text}
```

```
        onChange={(e) => setText(e.target.value)}
      />
      <button type="submit">Add</button>
    </form>
  );
};

export default AddTodo;
```

Now, your TODO application is ready. You can customize the styling in the `App.css` file.

Finally, to deploy it to GitHub Pages, follow these steps:

1. Create a GitHub repository for your project.

2. Initialize a git repository in your project folder if you haven't already (`git init`).

3. Add and commit your changes (`git add . && git commit -m "Initial commit"`).

4. Add your GitHub repository as a remote (`git remote add origin <your_repository_url>`).

5. Push your code to GitHub (`git push -u origin master`).

6. Install `gh-pages` package: `npm install gh-pages`.

7. Add a `homepage` field to your `package.json`:

```json
"homepage": "https://<username>.github.io/<repository_name>"
```

Replace `<username>` with your GitHub username and `<repository_name>` with the name of your GitHub repository.

8. Add scripts to deploy to GitHub Pages in your `package.json`:

```json
"scripts": {
  "predeploy": "npm run build",
  "deploy": "gh-pages -d build",
}
```

9. Run `npm run deploy` to deploy your application to GitHub Pages.

Now, your TODO application should be deployed and accessible at the URL you specified in the `homepage` field of your `package.json`.