Bachelor of Technology (B.Tech)

Department of Computer Science and Engineering

II year II sem- Skill Development Course
(Node JS/ React JS/Django)
Laboratory Manual

BALAJI INSTITUTE OF TECHNOLOGY AND SCIENCE (AUTONOMOUS)

B.Tech(Department of Computer Science & Engineering)

NODE JS/ REACT JS/ DJANGO

Course Outcomes: At the end of the course, the student will be able to,

- Build a custom website with HTML, CSS, and Bootstrap and little JavaScript.
- Demonstrate Advanced features of JavaScript and learn about JDBC.
- Develop Server – side implementation using Java technologies.
- Develop the server – side implementation using Node JS.
- Design a Single Page Application using React.

Lab Experiments:

1. Build a responsive shopping cart web app with registration, login, catalog, and cart pages using CSS3, flex, and grid.
2. Build a responsive shopping cart web app with registration, login, catalog, and cart pages using BootStrap Framework.
3. Use JavaScript for doing client – side validation of the pages implemented in experiment 1 and experiment 2.
4. Explore ES6 features like arrow functions, callbacks, promises, and async/await. Implement a web app to fetch weather data from OpenWeatherMap.org and display it as a graph.
5. Develop a java standalone application that connects with the database (Oracle/ MySQL) and perform the CRUD operation on the database tables.
6. Create an xml for the bookstore. Validate the same using both DTD and XSD.
7. Design a servlet-based controller to interact with the application from Experiment 1 and the database from Experiment 5.
8. Maintaining the transactional history of any user is very important. Explore the various session tracking mechanism (Cookies, HTTP Session).
9. Create a custom server using http module and explore the other modules of Node JS like OS, path, event.
10. Develop an express web application that can interact with REST API to perform CRUD operations on student data. (Use Postman)
11. For the above application create authorized end points using JWT (JSON Web Token).
12. Create a react application for the student management system having registration, login, contact, about pages and implement routing to navigate through these pages.
13. Create a React service to fetch weather data from OpenWeatherMap.org and display current and historical weather using Chart.js.
14. Create a TODO application in react with necessary components and deploy it into Github.

EXPERIMENT:1

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

Project Structure

HTML Files: Different pages for registration, login, catalog, and cart.

CSS Stylesheets: Main CSS file for styling.

JavaScript Files: For adding interactivity like cart management and user login/registration logic.

Backend Services: To manage user data and product catalog (you can use a simple Node.js/Express backend, but let's focus on the front-end in this guide).

Step 1: Set Up the HTML Structure

Start by creating the following HTML files:

index.html: The landing page and product catalog.

login.html: The user login page.

register.html: The user registration page.

cart.html: The shopping cart page.

Step 2: Create CSS Stylesheets

Create a styles.css file to hold the CSS rules for your application. Use Flexbox and Grid to ensure responsive layouts.

Step 3: Add HTML Content

Here's a simple outline for each of the pages with some key CSS styling features.

Catalog Page (index.html)

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Product Catalog</title>
```

```
    <link rel="stylesheet" href="styles.css">
</head>
<body>
    <header>
        <h1>Welcome to Our Shop</h1>
        <nav>
            <a href="index.html">Catalog</a>
            <a href="cart.html">Cart</a>
            <a href="login.html">Login</a>
        </nav>
    </header>

    <main>
        <div class="product-grid">
            <div class="product">
                <img src="example-product.jpg" alt="Product Image">
                <h3>Product Name</h3>
                <p>$19.99</p>
                <button>Add to Cart</button>
            </div>
            <!-- Add more product divs as needed -->
        </div>
    </main>

    <footer>
```

```
        <p>&copy; 2024 Our Shop. All rights reserved.</p>
    </footer>
</body>
</html>
```

Login Page (login.html)

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Login</title>
    <link rel="stylesheet" href="styles.css">
</head>
<body>
    <header>
        <h1>Login</h1>
        <nav>
            <a href="index.html">Catalog</a>
            <a href="cart.html">Cart</a>
        </nav>
    </header>

    <main>
        <form id="login-form">
            <label for="username">Username:</label>
            <input type="text" id="username" name="username">
```

```
            <label for="password">Password:</label>
            <input type="password" id="password" name="password">

            <button type="submit">Login</button>
        </form>
    </main>

    <footer>
        <p>&copy; 2024 Our Shop. All rights reserved.</p>
    </footer>
</body>
</html>
```

Registration Page (register.html)

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Register</title>
    <link rel="stylesheet" href="styles.css">
</head>
<body>
    <header>
        <h1>Register</h1>
        <nav>
```

```
            <a href="index.html">Catalog</a>
            <a href="cart.html">Cart</a>
        </nav>
    </header>

    <main>
        <form id="register-form">
            <label for="username">Username:</label>
            <input type="text" id="username" name="username">

            <label for="password">Password:</label>
            <input type="password" id="password" name="password">

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

    <footer>
        <p>&copy; 2024 Our Shop. All rights reserved.</p>
    </footer>
</body>
</html>
```

Cart Page (cart.html)

```
<!DOCTYPE html>
<html lang="en">
```

```
<head>
    <meta charset="UTF-8">
    <title>Shopping Cart</title>
    <link rel="stylesheet" href="styles.css">
</head>
<body>
    <header>
        <h1>Shopping Cart</h1>
        <nav>
            <a href="index.html">Catalog</a>
            <a href="login.html">Login</a>
        </nav>
    </header>

    <main>
        <div class="cart-items">
            <div class="cart-item">
                <h3>Product Name</h3>
                <p>Quantity: 1</p>
                <p>Price: $19.99</p>
            </div>
            <!-- Add more cart items as needed -->
        </div>

        <div class="cart-summary">
```

```
            <h3>Total: $19.99</h3>
            <button>Checkout</button>
        </div>
    </main>

    <footer>
        <p>&copy; 2024 Our Shop. All rights reserved.</p>
    </footer>
</body>
</html>
```

Step 4: Style with CSS

Here's an example of the CSS styling with Flexbox and Grid for a responsive layout (styles.css):

```
/* Basic Styles */
body {
    font-family: Arial, sans-serif;
    margin: 0;
    padding: 0;
}

header {
    background-color: #333;
    color: white;
    text-align: center;
    padding: 1em;
```

```
}

nav {
    display: flex;
    justify-content: center;
}

nav a {
    color: white;
    margin: 0 10px;
    text-decoration: none;
}

main {
    padding: 20px;
}

/* Product Grid */
.product-grid {
    display: grid;
    grid-template-columns: repeat(auto-fill, minmax(200px, 1fr));
    gap: 20px;
}

.product {
```

```
    border: 1px solid #ddd;
    padding: 10px;
    text-align: center;
}

.product img {
    max-width: 100%;
}

.product button {
    background-color: #333;
    color: white;
    border: none;
    padding: 10px;
    cursor: pointer;
}

.product button:hover {
    background-color: #555;
}

/* Cart */
.cart-items {
    display: flex;
    flex-direction: column;
```

```
}

.cart-item {
    border-bottom: 1px solid #ddd;
    padding: 10px;
}

.cart-summary {
    text-align: right;
    padding: 10px;
}

.cart-summary button {
    background-color: #333;
    color: white;
    border: none;
    padding: 10px;
    cursor: pointer;
}

.cart-summary button:hover {
    background-color: #555;
}

/* Forms */
```

```
form {
    max-width: 400px;
    margin: 0 auto;
}

label {
    display: block;
    margin-bottom: 10px;
}

input {
    width: 100%;
    padding: 10px;
    border: 1px solid #ddd;
    margin-bottom: 20px;
}

button {
    display: block;
    width: 100%;
    background-color: #333;
    color: white;
    padding: 10px;
    border: none;
    cursor: pointer;
```

```
}

button:hover {
    background-color: #555;
}

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

Step 5: Add JavaScript Logic

For simplicity, here's an example of JavaScript logic for adding items to the cart and handling login/registration (app.js):

```
document.addEventListener("DOMContentLoaded", () => {
    // Add event listeners for adding to cart
    const addCartButtons = document.querySelectorAll(".product button");
    addCartButtons.forEach((button) => {
        button.addEventListener("click", () => {
            // Logic to add to cart
            console.log("Item added to cart");
```

```
        });
    });

    // Event listener for login form
    const loginForm = document.getElementById("login-form");
    if (loginForm) {
        loginForm.addEventListener("submit", (e) => {
            e.preventDefault();
            // Login logic
            console.log("User logged in");
        });
    }

    // Event listener for registration form
    const registerForm = document.getElementById("register-form");
    if (registerForm) {
        registerForm.addEventListener("submit", (e) => {
            e.preventDefault();
            // Registration logic
            console.log("User registered");
        });
    }
});
```

EXPERIMENT: 2

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

Project Structure
HTML Files: Registration, login, catalog, and cart pages.
CSS Stylesheets: Main CSS file for additional custom styles.
JavaScript Files: Logic for interactivity like managing the cart and user authentication.
Bootstrap Assets: Include Bootstrap's CSS and JavaScript for styling and functionality.

Step 1: Include Bootstrap Framework
In your HTML files, include the Bootstrap CSS and JavaScript files. You can use a CDN for convenience:

```
<!-- Include Bootstrap CSS -->
<link rel="stylesheet"
href="https://stackpath.bootstrapcdn.com/bootstrap/4.5.2/css/bootstrap.min.css">

<!-- Include jQuery, Popper.js, and Bootstrap JavaScript -->
<script src="https://code.jquery.com/jquery-3.5.1.slim.min.js"></script>
<script
src="https://cdn.jsdelivr.net/npm/@popperjs/core@2.11.6/dist/umd/popper.min.js"></scr
ipt>
<script
src="https://stackpath.bootstrapcdn.com/bootstrap/4.5.2/js/bootstrap.min.js"></script>
```

Step 2: Create HTML Pages with Bootstrap
Now let's create the pages with Bootstrap. We'll use the grid system, components like cards, forms, and utilities to build a responsive design.

Catalog Page (index.html)

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Product Catalog</title>
```

```
    <link rel="stylesheet" href="styles.css">
    <!-- Include Bootstrap -->
    <link rel="stylesheet"
href="https://stackpath.bootstrapcdn.com/bootstrap/4.5.2/css/bootstrap.min.css">
</head>
<body>
    <nav class="navbar navbar-expand-lg navbar-dark bg-dark">
        <a class="navbar-brand" href="#">Our Shop</a>
        <button class="navbar-toggler" type="button" data-toggle="collapse" data-
target="#navbarNav" aria-controls="navbarNav" aria-expanded="false" aria-
label="Toggle navigation">
            <span class="navbar-toggler-icon"></span>
        </button>
        <div class="collapse navbar-collapse" id="navbarNav">
            <ul class="navbar-nav ml-auto">
                <li class="nav-item">
                    <a class="nav-link" href="index.html">Catalog</a>
                </li>
                <li class="nav-item">
                    <a class="nav-link" href="cart.html">Cart</a>
                </li>
                <li class="nav-item">
                    <a class="nav-link" href="login.html">Login</a>
                </li>
            </ul>
        </div>
    </nav>

    <div class="container mt-4">
        <div class="row">
            <!-- Product Card -->
            <div class="col-md-4">
                <div class="card">
                    <img src="example-product.jpg" class="card-img-top" alt="Product
Image">
                    <div class="card-body">
                        <h5 class="card-title">Product Name</h5>
                        <p class="card-text">$19.99</p>
                        <button class="btn btn-primary">Add to Cart</button>
                    </div>
```

```
            </div>
          </div>
          <!-- Add more product cards as needed -->
        </div>
    </div>

    <footer class="bg-dark text-white text-center py-3 mt-4">
        <p>&copy; 2024 Our Shop. All rights reserved.</p>
    </footer>

    <!-- Include Bootstrap JavaScript -->
    <script src="https://code.jquery.com/jquery-3.5.1.slim.min.js"></script>
    <script
src="https://cdn.jsdelivr.net/npm/@popperjs/core@2.11.6/dist/umd/popper.min.js"></scr
ipt>
    <script
src="https://stackpath.bootstrapcdn.com/bootstrap/4.5.2/js/bootstrap.min.js"></script>
</body>
</html>
```

Login Page (login.html)

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Login</title>
    <link rel="stylesheet" href="styles.css">
    <link rel="stylesheet"
href="https://stackpath.bootstrapcdn.com/bootstrap/4.5.2/css/bootstrap.min.css">
</head>
<body>
    <nav class="navbar navbar-expand-lg navbar-dark bg-dark">
        <a class="navbar-brand" href="#">Our Shop</a>
        <div class="collapse navbar-collapse">
            <ul class="navbar-nav ml-auto">
                <li class="nav-item">
                    <a class="nav-link" href="index.html">Catalog</a>
                </li>
                <li class="nav-item">
                    <a class="nav-link" href="cart.html">Cart</a>
                </li>
```

```
            </ul>
        </div>
    </nav>

    <div class="container mt-4">
        <h2>Login</h2>
        <form id="login-form">
            <div class="form-group">
                <label for="username">Username:</label>
                <input type="text" class="form-control" id="username" name="username">
            </div>
            <div class="form-group">
                <label for="password">Password:</label>
                <input type="password" class="form-control" id="password"
name="password">
            </div>
            <button type="submit" class="btn btn-primary">Login</button>
        </form>
    </div>

    <footer class="bg-dark text-white text-center py-3 mt-4">
        <p>&copy; 2024 Our Shop. All rights reserved.</p>
    </footer>

    <!-- Include Bootstrap JavaScript -->
    <script src="https://code.jquery.com/jquery-3.5.1.slim.min.js"></script>
    <script
src="https://cdn.jsdelivr.net/npm/@popperjs/core@2.11.6/dist/umd/popper.min.js"></scr
ipt>
    <script
src="https://stackpath.bootstrapcdn.com/bootstrap/4.5.2/js/bootstrap.min.js"></script>
</body>
</html>
```

Registration Page (register.html)

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Register</title>
    <link rel="stylesheet" href="styles.css">
```

```
    <link rel="stylesheet"
href="https://stackpath.bootstrapcdn.com/bootstrap/4.5.2/css/bootstrap.min.css">
</head>
<body>
    <nav class="navbar navbar-expand-lg navbar-dark bg-dark">
        <a class="navbar-brand" href="#">Our Shop</a>
        <div class="collapse navbar-collapse">
            <ul class="navbar-nav ml-auto">
                <li class="nav-item">
                    <a class="nav-link" href="index.html">Catalog</a>
                </li>
                <li class="nav-item">
                    <a class="nav-link" href="cart.html">Cart</a>
                </li>
            </ul>
        </div>
    </nav>

    <div class="container mt-4">
        <h2>Register</h2>
        <form id="register-form">
            <div class="form-group">
                <label for="username">Username:</label>
                <input type="text" class="form-control" id="username" name="username">
            </div>
            <div class="form-group">
                <label for="password">Password:</label>
                <input type="password" class="form-control" id="password"
name="password">
            </div>
            <button type="submit" class="btn btn-primary">Register</button>
        </form>
    </div>

    <footer class="bg-dark text-white text-center py-3 mt-4">
        <p>&copy; 2024 Our Shop. All rights reserved.</p>
    </footer>

    <!-- Include Bootstrap JavaScript -->
    <script src="https://code.jquery.com/jquery-3.5.1.slim.min.js"></script>
```

```
    <script
src="https://cdn.jsdelivr.net/npm/@popperjs/core@2.11.6/dist/umd/popper.min.js"></scr
ipt>
    <script
src="https://stackpath.bootstrapcdn.com/bootstrap/4.5.2/js/bootstrap.min.js"></script>
</body>
</html>
```

Cart Page (cart.html)

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Shopping Cart</title>
    <link rel="stylesheet" href="styles.css">
    <link rel="stylesheet"
href="https://stackpath.bootstrapcdn.com/bootstrap/4.5.2/css/bootstrap.min.css">
</head>
<body>
    <nav class="navbar navbar-expand-lg navbar-dark bg-dark">
        <a class="navbar-brand" href="#">Our Shop</a>
        <div class="collapse navbar-collapse">
            <ul class="navbar-nav ml-auto">
                <li class="nav-item">
                    <a class="nav-link" href="index.html">Catalog</a>
                </li>
                <li class="nav-item">
                    <a class="nav-link" href="login.html">Login</a>
                </
```

EXPERIMENT:3

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

client-side validation helps ensure that user input meets certain criteria before it's sent to the server. This reduces the number of server-side validation checks and can provide a better user experience by alerting users to errors early.

Here's an outline to implement JavaScript-based client-side validation for a shopping cart application with registration, login, catalog, and cart pages. We'll focus on validation for the registration and login forms.

Validation Checks
Common validation checks include:

Required fields: Ensuring certain fields are not left blank.
Minimum/maximum length: For example, a username should have a minimum and maximum length.
Email format: Validating if an email address is in the correct format.
Password complexity: Checking if the password meets complexity requirements (e.g., contains a mix of characters, numbers, symbols, etc.).
Let's use the registration and login pages from our previous response and add JavaScript-based validation logic.

Registration Page (register.html)

```
<!DOCTYPE html>
<html lang="en">
<head>
   <meta charset="UTF-8">
   <title>Register</title>
   <link                                                                rel="stylesheet"
href="https://stackpath.bootstrapcdn.com/bootstrap/4.5.2/css/bootstrap.min.css">
</head>
<body>
   <nav class="navbar navbar-expand-lg navbar-dark bg-dark">
      <a class="navbar-brand" href="#">Our Shop</a>
      <div class="collapse navbar-collapse">
         <ul class="navbar-nav ml-auto">
            <li class="nav-item">
```

```
                <a class="nav-link" href="index.html">Catalog</a>
            </li>
            <li class="nav-item">
                <a class="nav-link" href="cart.html">Cart</a>
            </li>
        </ul>
    </div>
</nav>

<div class="container mt-4">
    <h2>Register</h2>
    <form id="register-form">
        <div class="form-group">
            <label for="username">Username:</label>
            <input type="text" class="form-control" id="username" name="username" required>
            <small class="form-text text-muted">Username must be between 3 and 20 characters long.</small>
        </div>
        <div class="form-group">
            <label for="email">Email:</label>
            <input type="email" class="form-control" id="email" name="email" required>
        </div>
        <div class="form-group">
            <label for="password">Password:</label>
            <input type="password" class="form-control" id="password" name="password" required>
            <small class="form-text text-muted">Password must contain at least 8 characters, including at least one uppercase letter, one lowercase letter, and one number.</small>
        </div>
        <button type="submit" class="btn btn-primary">Register</button>
    </form>
</div>

<footer class="bg-dark text-white text-center py-3 mt-4">
    <p>&copy; 2024 Our Shop. All rights reserved.</p>
</footer>

<!-- JavaScript for client-side validation -->
```

```
    <script>
        document.getElementById("register-form").addEventListener("submit",
function(event) {
            const username = document.getElementById("username").value;
            const email = document.getElementById("email").value;
            const password = document.getElementById("password").value;

            let errorMessage = "";

            // Username validation
            if (username.length < 3 || username.length > 20) {
                errorMessage += "Username must be between 3 and 20 characters long. ";
            }

            // Email validation
            const emailPattern = /^[^\s@]+@[^\s@]+\.[^\s@]+$/;
            if (!emailPattern.test(email)) {
                errorMessage += "Invalid email format. ";
            }

            // Password validation
            const passwordPattern = /^(?=.*[a-z])(?=.*[A-Z])(?=.*\d)[A-Za-z\d]{8,}$/;
            if (!passwordPattern.test(password)) {
                errorMessage += "Password must contain at least one uppercase letter, one
lowercase letter, and one number, and be at least 8 characters long. ";
            }

            if (errorMessage) {
                alert(errorMessage);
                event.preventDefault(); // Prevent form submission if validation fails
            }
        });
    </script>

    <!-- Include Bootstrap JavaScript -->
    <script src="https://code.jquery.com/jquery-3.5.1.slim.min.js"></script>
    <script
src="https://cdn.jsdelivr.net/npm/@popperjs/core@2.11.6/dist/umd/popper.min.js"></scr
ipt>
```

```
    <script
src="https://stackpath.bootstrapcdn.com/bootstrap/4.5.2/js/bootstrap.min.js"></script>
</body>
</html>
```

This script validates the username, email, and password fields according to the specified criteria. If validation fails, it prevents the form from being submitted and displays an alert with the validation errors.

Login Page (login.html):

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Login</title>
    <link rel="stylesheet"
href="https://stackpath.bootstrapcdn.com/bootstrap/4.5.2/css/bootstrap.min.css">
</head>
<body>
    <nav class="navbar navbar-expand-lg navbar-dark bg-dark">
        <a class="navbar-brand" href="#">Our Shop</a>
        <div class="collapse navbar-collapse">
            <ul class="navbar-nav ml-auto">
                <li class="nav-item">
                    <a class="nav-link" href="index.html">Catalog</a>
                </li>
                <li class="nav-item">
                    <a class="nav-link" href="cart.html">Cart</a>
                </li>
            </ul>
        </div>
    </nav>

    <div class="container mt-4">
        <h2>Login</h2>
        <form id="login-form">
            <div class="form-group">
                <label for="username">Username:</label>
                <input type="text" class="form-control" id="username" name="username"
required>
            </div>
```

```
            <div class="form-group">
                <label for="password">Password:</label>
                <input type="password" class="form-control" id="password"
name="password" required>
            </div>
            <button type="submit" class="btn btn-primary">Login</button>
        </form>
    </div>

    <footer class="bg-dark text-white text-center py-3 mt-4">
        <p>&copy; 2024 Our Shop. All rights reserved.</p>
    </footer>

    <!-- JavaScript for client-side validation -->
    <script>
        document.getElementById("login-form").addEventListener("submit",
function(event) {
            const username = document.getElementById("username").value;
            const password = document.getElementById("password").value;

            let errorMessage = "";

            // Check if username and password are not empty
            if (username.trim() === "") {
                errorMessage += "Username is required. ";
            }

            if (password.trim() === "") {
                errorMessage += "Password is required. ";
            }

            if (errorMessage) {
                alert(errorMessage);
                event.preventDefault(); // Prevent form submission if validation fails
            }
        });
    </script>

    <!-- Include Bootstrap JavaScript -->
    <script src="https://code.jquery.com/jquery-3.5.1.slim.min.js"></script>
```

```
    <script
src="https://cdn.jsdelivr.net/npm/@popperjs/core@2.11.6/dist/umd/popper.min.js"></scr
ipt>
  <script
src="https://stackpath.bootstrapcdn.com/bootstrap/4.5.2/js/bootstrap.min.js"></script>
</body>
</html>
```

In this script, the validation checks if the username and password fields are not empty before submitting the form. If the validation fails, an alert message is shown, and the form submission is prevented.

Conclusion

With the above scripts, you've added basic client-side validation to your registration and login pages. These validations ensure that users provide the correct information before submitting their forms. This doesn't replace server-side validation but complements it to improve user experience and reduce unnecessary server load.

EXPERIMENT: 4

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 create a weather information application that retrieves weather data from OpenWeatherMap and displays it in the form of a graph, you can leverage ES6 features like arrow functions, callbacks, promises, and async/await. While I don't have internet access to connect to OpenWeatherMap or retrieve live weather data, I can guide you through the steps to implement this application in a realistic setting.

The implementation involves the following key steps:

Set up OpenWeatherMap: Sign up for an API key at OpenWeatherMap.
Fetch Weather Data: Use Fetch API with promises and async/await to get weather data.
Plot the Graph: Use a JavaScript library like Chart.js or D3.js to create the graph.
Use ES6 Features: Apply arrow functions, callbacks, promises, and async/await.
Let's break down the implementation:

Step 1: Set Up OpenWeatherMap
Sign up at OpenWeatherMap and obtain your API key.
Determine which weather data you want to retrieve (e.g., current weather, forecast).
Step 2: Fetch Weather Data
Use the Fetch API with promises and async/await to get weather data from OpenWeatherMap. Here's an example using promises:

```
const apiKey = "YOUR_API_KEY"; // Replace with your OpenWeatherMap API key
const city = "London"; // Example city
const url = `https://api.openweathermap.org/data/2.5/weather?q=${city}&appid=${apiKey}`;

// Fetch weather data using Promises
fetch(url)
  .then(response => response.json())
  .then(data => {
    console.log("Weather Data:", data);
    // Process and use the weather data
  })
```

```
.catch(error => {
    console.error("Error fetching weather data:", error);
});
```

Step 3: Fetch Data with Async/Await

The async/await syntax is more straightforward and helps with readability when dealing with promises. Here's the same example using async/await:

```
const apiKey = "YOUR_API_KEY";
const city = "London";
const url =
`https://api.openweathermap.org/data/2.5/weather?q=${city}&appid=${apiKey}`;

async function fetchWeatherData() { try
  {
    const response = await fetch(url); const
    data = await response.json();
    console.log("Weather Data:", data);
    // Process and use the weather data
  } catch (error) {
    console.error("Error fetching weather data:", error);
  }
}

// Call the function to fetch the weather data
fetchWeatherData();
```

Step 4: Display Data on a Web Page

To display weather information as a graph, use a charting library like Chart.js. Here's an example that fetches weather data and plots a simple graph with temperature data:

Html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Weather Information</title>
  <script src="https://cdn.jsdelivr.net/npm/chart.js"></script>
</head>
```

```
<body>
    <h2>Weather Information</h2>
    <canvas id="weather-chart" width="400" height="200"></canvas>

    <script>
        const apiKey = "YOUR_API_KEY";
        const city = "London";
        const url =
`https://api.openweathermap.org/data/2.5/weather?q=${city}&appid=${apiKey}`;

        async function fetchWeatherData() {
        try {
                const response = await fetch(url); const
                data = await response.json();

                const tempK = data.main.temp; // Temperature in Kelvin const
                tempC = tempK - 273.15; // Convert to Celsius

                const ctx = document.getElementById("weather-
chart").getContext("2d");
                const weatherChart = new Chart(ctx, {
                    type: "bar",
                    data: {
                        labels: ["Temperature"],
                        datasets: [{
                            label: `Temperature in ${city} (°C)`,
                            data: [tempC],
                            backgroundColor: "rgba(75, 192, 192, 0.6)",
                            borderColor: "rgba(75, 192, 192, 1)",
                            borderWidth: 1
                        }]
                    },
                    options: {
                        scales: {
                            yAxes: [{
                                ticks: {
                                    beginAtZero: true
                                }
                            }]
                        }
```

```
            }
        });

    } catch (error) {
        console.error("Error fetching weather data:", error);
    }
}

// Call the function to fetch the weather data
fetchWeatherData();
</script>
</body>
</html>
```

This code snippet fetches weather data from OpenWeatherMap and plots a simple bar chart showing the current temperature in Celsius. You can use this as a starting point and expand it to display other weather parameters or use additional Chart.js features for more complex graphs.

Conclusion

This example demonstrates how to implement an application to fetch weather information and plot it on a graph using ES6 features like arrow functions, callbacks, promises, and async/await. By following this guide, you can build a functional weather information application with client-side visualization.

EXPERIMENT: 5

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

To create a standalone Java application that connects with a database (Oracle/MySQL) and performs CRUD (Create, Read, Update, Delete) operations, you need to do the following:

Set up a Java Development Environment: Ensure you have Java Development Kit (JDK) and an Integrated Development Environment (IDE) like Eclipse or IntelliJ IDEA.
Install Database and Set up a Database Connection: Set up a local or remote Oracle or MySQL database and obtain connection details (like username, password, and database URL).
Add JDBC Driver: Download and add the appropriate Java Database Connectivity (JDBC) driver for the database to your project.
Implement CRUD Operations: Write Java code to connect to the database and perform CRUD operations.
Handle Database Connection and Resource Management: Ensure proper handling of connections and resources to prevent memory leaks.
Let's focus on a MySQL example to demonstrate the CRUD operations in a Java standalone application.

Prerequisites
Java Development Kit (JDK)
Integrated Development Environment (IDE)
MySQL Database: Installed on your system or accessible remotely.
MySQL JDBC Driver: Obtain from MySQL Connector/J.
Step 1: Set Up MySQL Database
Install MySQL and set up a database. For this example, create a database called sample_db.
Create a table called employees with the following fields:
id (Primary Key, AUTO_INCREMENT)
name (VARCHAR(100))
age (INT)
department (VARCHAR(50))

CREATE DATABASE sample_db;

USE sample_db;

```
CREATE TABLE employees (
  id INT AUTO_INCREMENT,
  name VARCHAR(100),
  age INT,
  department VARCHAR(50),
  PRIMARY KEY (id)
);
```

Step 2: Add MySQL JDBC Driver to the Project

Add the MySQL JDBC driver (Connector/J) to your project. If you're using an IDE, you can import it as a Maven/Gradle dependency or manually add the JAR file to your project.

For Maven, add this dependency to your pom.xml:

```
<dependency>
  <groupId>mysql</groupId>
  <artifactId>mysql-connector-java</artifactId>
  <version>8.0.29</version>
</dependency>
```

Step 3: Implement CRUD Operations

Create a Java class that connects to the MySQL database and performs CRUD operations.

```
import java.sql.*;

public class EmployeeDatabase {
    private static final String DB_URL = "jdbc:mysql://localhost:3306/sample_db";
    private static final String DB_USER = "root"; // Replace with your DB username
    private static final String DB_PASSWORD = "password"; // Replace with your DB password

    // Connect to the database
    private static Connection connect() throws SQLException {
        return DriverManager.getConnection(DB_URL, DB_USER, DB_PASSWORD);
    }

    // Create a new employee record
    public static void createEmployee(String name, int age, String department) {
        String sql = "INSERT INTO employees (name, age, department) VALUES (?, ?, ?)";
```

```
        try (Connection conn = connect(); PreparedStatement pstmt =
conn.prepareStatement(sql)) {
            pstmt.setString(1, name);
            pstmt.setInt(2, age);
            pstmt.setString(3, department);
            pstmt.executeUpdate();
            System.out.println("Employee record created.");
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }

    // Read all employee records
    public static void readEmployees() {
        String sql = "SELECT * FROM employees";
        try (Connection conn = connect(); Statement stmt = conn.createStatement();
             ResultSet rs = stmt.executeQuery(sql)) {
            while (rs.next()) {
                System.out.println("ID: " + rs.getInt("id") +
                                   ", Name: " + rs.getString("name") +
                                   ", Age: " + rs.getInt("age") +
                                   ", Department: " + rs.getString("department"));
            }
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }

    // Update an existing employee record
    public static void updateEmployee(int id, String name, int age, String department) {
        String sql = "UPDATE employees SET name = ?, age = ?, department = ? WHERE
id = ?";
        try (Connection conn = connect(); PreparedStatement pstmt =
conn.prepareStatement(sql)) {
            pstmt.setString(1, name);
            pstmt.setInt(2, age);
            pstmt.setString(3, department);
            pstmt.setInt(4, id);
            pstmt.executeUpdate();
            System.out.println("Employee record updated.");
```

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

    // Delete an employee record
    public static void deleteEmployee(int id) {
        String sql = "DELETE FROM employees WHERE id = ?";
        try (Connection conn = connect(); PreparedStatement pstmt =
conn.prepareStatement(sql)) {
            pstmt.setInt(1, id);
            pstmt.executeUpdate();
            System.out.println("Employee record deleted.");
        } catch (SQLException e) {
            e.printStacktrace();
        }
    }

    public static void main(String[] args) {
        // Create an employee
        createEmployee("John Doe", 30, "Engineering");

        // Read all employees
        readEmployees();

        // Update an employee
        updateEmployee(1, "Jane Doe", 32, "Human Resources");

        // Delete an employee
        deleteEmployee(1);
    }
}
```

Explanation

connect(): Establishes a connection with the MySQL database using DriverManager.getConnection().
createEmployee(): Inserts a new record into the employees table.
readEmployees(): Reads and prints all records from the employees table.
updateEmployee(): Updates an existing record based on the provided id.
deleteEmployee(): Deletes a record with the specified id.

Conclusion

With the given example, you've created a standalone Java application that connects with a MySQL database and performs CRUD operations. This code structure should be a solid foundation for database interactions and resource management in Java. If you're working with Oracle, the approach is similar, with minor adjustments for the JDBC driver and connection details.

EXPERIMENT: 6

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

Creating an XML for a bookstore involves defining the structure of the XML file to represent information about books, authors, and related metadata. To ensure the XML adheres to a specific structure, you can validate it using Document Type Definition (DTD) or XML Schema Definition (XSD). DTD is simpler but less expressive, while XSD is more robust and supports data types and namespaces.

Let's start by creating an XML file representing a simple bookstore, then validate it with both DTD and XSD.

```
<?xml version="1.0" encoding="UTF-8"?>
<bookstore>
   <book isbn="978-3-16-148410-0">
      <title>Effective Java</title>
      <author>Joshua Bloch</author>
      <price>45.0</price>
   </book>
   <book isbn="978-0-596-52068-7">
      <title>Head First Java</title>
      <author>Kathy Sierra</author>
      <price>30.0</price>
   </book>
</bookstore>
```

Step 2: Create DTD for Validation

To validate the bookstore XML with a DTD, define the expected structure. The following DTD specifies the required elements, their order, and attributes.

```
<!DOCTYPE bookstore [
   <!ELEMENT bookstore (book+)>
   <!ELEMENT book (title, author, price)>
   <!ATTLIST book
      isbn CDATA #REQUIRED
   >
   <!ELEMENT title (#PCDATA)>
   <!ELEMENT author (#PCDATA)>
   <!ELEMENT price (#PCDATA)>
]>
```

This DTD specifies:

<bookstore> contains one or more <book> elements.
<book> contains <title>, <author>, and <price>.
The isbn attribute of <book> is required.
To validate the XML with this DTD, add a reference to it in the XML file:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE bookstore SYSTEM "bookstore.dtd">
<bookstore>
  <book isbn="978-3-16-148410-0">
    <title>Effective Java</title>
    <author>Joshua Bloch</author>
    <price>45.0</price>
  </book>
  <book isbn="978-0-596-52068-7">
    <title>Head First Java</title>
    <author>Kathy Sierra</author>
    <price>30.0</price>
  </book>
</bookstore>
```

Step 3: Create XSD for Validation
An XSD defines a more complex schema with data types, constraints, and namespaces.
Here's an XSD to validate the bookstore XML:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="bookstore">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="book" minOccurs="1" maxOccurs="unbounded">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="title" type="xs:string"/>
              <xs:element name="author" type="xs:string"/>
              <xs:element name="price" type="xs:float"/>
            </xs:sequence>
            <xs:attribute name="isbn" type="xs:string" use="required"/>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
```

```
      </xs:complexType>
   </xs:element>
</xs:schema>
```

This XSD schema specifies:

<bookstore> contains one or more <book> elements.
<book> has isbn as a required attribute and contains <title>, <author>, and <price>.
The data types are explicitly defined (string, float).
To validate the XML with this XSD, add a reference to it in the XML file:

```
<?xml version="1.0" encoding="UTF-8"?>
<bookstore xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:noNamespaceSchemaLocation="bookstore.xsd">
   <book isbn="978-3-16-148410-0">
      <title>Effective Java</title>
      <author>Joshua Bloch</author>
      <price>45.0</price>
   </book>
   <book isbn="978-0-596-52068-7">
      <title>Head First Java</title>
      <author>Kathy Sierra</author>
      <price>30.0</price>
   </book>
</bookstore>
```

Conclusion

This example demonstrates creating an XML file for a bookstore and validating it using both DTD and XSD. While DTD is suitable for simple validation, XSD offers more complex schema definitions with data types and additional constraints. By referencing the DTD or XSD in the XML file, you ensure the XML structure aligns with the defined schema, providing validation and error-checking.

EXPERIMENT: 7

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

To design a Java-based controller with servlets to interact with a specific application and a database. Here's a general outline for implementing a servlet-based controller in a Java web application with a database connection. This is typically done using the Servlet API, JDBC, and a Java EE framework like Spring.

Step 1: Set Up Servlet

First, let's set up a basic servlet that will handle HTTP requests and responses.

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

public class MyServlet extends HttpServlet {
   @Override
   protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws
ServletException, IOException {
      resp.setContentType("text/html");
      resp.getWriter().println("This is the response from GET method.");
   }

   @Override
   protected void doPost(HttpServletRequest req, HttpServletResponse resp) throws
ServletException, IOException {
      resp.setContentType("text/html");
      resp.getWriter().println("This is the response from POST method.");
   }
}
```

This servlet listens for HTTP GET and POST requests and sends a simple response. Now, let's see how to interact with a database.

Step 2: Database Connection with JDBC

Next, you need to connect to your database. Make sure you have the JDBC driver for your database and include it in your project's dependencies.

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
```

```
public class DatabaseConnection {
   private static final String DB_URL = "jdbc:mysql://localhost:3306/mydatabase"; //
Example for MySQL
   private static final String USER = "myuser";
   private static final String PASS = "mypassword";

   public static Connection getConnection() throws SQLException {
      return DriverManager.getConnection(DB_URL, USER, PASS);
   }
}
```

Step 3: Integrate Database Interaction in Servlet

With the servlet and database connection in place, you can now create a controller that interacts with the database. For example, let's create a servlet that retrieves a list of users from the database.

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

public class UserController extends HttpServlet {
   @Override
   protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws
ServletException, IOException {
      resp.setContentType("text/html");
      PrintWriter out = resp.getWriter();

      try (Connection conn = DatabaseConnection.getConnection()) {
         Statement stmt = conn.createStatement();
         ResultSet rs = stmt.executeQuery("SELECT * FROM users");

         out.println("<h1>List of Users:</h1>");
         while (rs.next()) {
            out.println("<p>" + rs.getString("name") + " - " + rs.getString("email") +
"</p>");
         }

      } catch (SQLException e) {
         e.printStackTrace();
         out.println("Error interacting with the database.");
```

```
        }
    }
}
```

Step 3: Integrate Database Interaction in Servlet

With the servlet and database connection in place, you can now create a controller that interacts with the database. For example, let's create a servlet that retrieves a list of users from the database.

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

public class UserController extends HttpServlet {
    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws
ServletException, IOException {
        resp.setContentType("text/html");
        PrintWriter out = resp.getWriter();

        try (Connection conn = DatabaseConnection.getConnection()) {
            Statement stmt = conn.createStatement();
            ResultSet rs = stmt.executeQuery("SELECT * FROM users");

            out.println("<h1>List of Users:</h1>");
            while (rs.next()) {
                out.println("<p>" + rs.getString("name") + " - " + rs.getString("email") +
"</p>");
            }

        } catch (SQLException e) {
            e.printStackTrace();
            out.println("Error interacting with the database.");
        }
    }
}
```

This servlet fetches a list of users from the database and displays it as an HTML response. You can modify the SQL query and the servlet logic to suit your specific requirements.

Step 4: Web Deployment and Configuration
To deploy this servlet, you need to set up a web application and configure the deployment descriptor (web.xml). Here's an example web.xml configuration:

```
<web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
http://xmlns.jcp.org/xml/ns/javaee/web-app_4_0.xsd" version="4.0">

   <servlet>
      <servlet-name>UserController</servlet-name>
      <servlet-class>UserController</servlet-class>
   </servlet>

   <servlet-mapping>
      <servlet-name>UserController</servlet-name>
      <url-pattern>/users</url-pattern>
   </servlet-mapping>

</web-app>
```

This web.xml configuration maps the UserController servlet to the /users URL pattern. To deploy the web application, use a servlet container like Apache Tomcat, which can be configured to read and execute the web.xml file.

Conclusion
This guide provides a basic structure for implementing a servlet-based controller that interacts with a database. Depending on your specific requirements, you might need to adjust the servlet's logic, database queries, and other configurations. Additionally, consider using frameworks like Spring for more robust and scalable solutions.

EXPERIMENT: 8

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

Session tracking is an essential concept in web development, allowing web applications to maintain user data across multiple HTTP requests. Since HTTP is stateless, different mechanisms are used to create a sense of continuity or "state" across user interactions with a website. Here's an overview of the two primary session tracking mechanisms—Cookies and HTTP Sessions—along with other related techniques:

Cookies

Cookies are small pieces of data stored on the client-side by the web browser. They can be used to track user sessions, store user preferences, and maintain other user-specific information.

Creation and Retrieval: Cookies are created by the server and sent to the client via HTTP response headers (Set-Cookie). When the client makes subsequent requests, it sends the cookies back to the server, allowing the server to retrieve session-related information.
Attributes: Cookies can have various attributes like name, value, domain, path, expires, max-age, and secure. These attributes control the scope, lifespan, and security of cookies.
Use Cases: Cookies are commonly used for session tracking, authentication, user preferences, and tracking website analytics.
Security Considerations: Cookies can be vulnerable to attacks such as cross-site scripting (XSS) and cross-site request forgery (CSRF). To mitigate these risks, attributes like HttpOnly, Secure, and SameSite can be used to enhance cookie security.
HTTP Sessions
HTTP Sessions are server-side constructs that maintain state for a user during a web session. They are typically used to track users between requests on the server side.

Session Management: When a client initiates a session, the server creates a session object and assigns a unique session ID. This session ID is often stored in a cookie (JSESSIONID for Java applications) or passed as a URL parameter.
Session Scope: HTTP sessions are generally user-specific, maintaining information such as authentication status, user roles, shopping cart contents, or other application-specific data.
Session Lifetime: Sessions have a specific lifespan, which can be set by the server. If a session remains inactive beyond its lifespan, it is invalidated, and the user must re-authenticate or start a new session.
Security Considerations: While HTTP sessions are stored server-side, session IDs need to be protected. Common security practices include using secure cookies, HTTPS,

session regeneration after login, and setting appropriate timeouts to prevent session hijacking.

Other Session Tracking Mechanisms

Besides Cookies and HTTP Sessions, there are other mechanisms for session tracking:

Hidden Form Fields: Storing session information in hidden form fields allows state to persist across form submissions. However, this approach is less secure and prone to tampering.

URL Rewriting: Embedding session IDs in URLs allows session tracking without cookies, but it has security risks (e.g., exposing session IDs in logs, bookmarks, or shared URLs).

Local Storage/Session Storage: Modern browsers support local storage and session storage, allowing web applications to maintain state on the client side. While useful for certain cases, these mechanisms are typically not used for session tracking in the context of user authentication.

Conclusion

The choice between cookies and HTTP sessions depends on the specific requirements of your application. Cookies are useful for lightweight client-side state management, while HTTP sessions are ideal for server-side session tracking and authentication. To ensure security, implement best practices like HTTPS, secure cookies, and proper session management to prevent common security vulnerabilities.

EXPERIMENT: 9

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

Creating a custom server with Node.js involves using the built-in http module to set up a basic HTTP server. Node.js also provides several other modules, like os for system information, path for working with file paths, and events for handling events. Here's a guide to setting up a simple HTTP server and exploring these other modules:

Step 1: Set Up a Basic HTTP Server

The http module in Node.js allows you to create a simple HTTP server that listens for incoming requests and sends responses.

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

const server = http.createServer((req, res) => {
   res.writeHead(200, { 'Content-Type': 'text/plain' });
   res.write('Hello, World!');
   res.end();
});

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

This basic server listens on port 3000 and responds with "Hello, World!" to any incoming HTTP request. You can change the content type and response message as needed.

Step 2: Use the OS Module

The os module provides information about the operating system. Here's an example that retrieves some system information and displays it on a webpage.

```
const os = require('os');

server.on('request', (req, res) => {
   if (req.url === '/system-info') {
      const systemInfo = {
         platform: os.platform(),
         arch: os.arch(),
         uptime: os.uptime(),
         freeMem: os.freemem(),
```

```
            totalMem: os.totalmem(),
            cpus: os.cpus(),
        };

        res.writeHead(200, { 'Content-Type': 'application/json' });
        res.write(JSON.stringify(systemInfo));
        res.end();
    }
});
```

Step 3: Use the Path Module

The path module provides utilities for working with file and directory paths. Here's an example that retrieves the file name from a given path.

```
Javascript
const path = require('path');

server.on('request', (req, res) => {
    if (req.url === '/file-info') {
        const filePath = __filename; // Current file path
        const fileName = path.basename(filePath); // Get the file name from the path

        res.writeHead(200, { 'Content-Type': 'text/plain' });
        res.write(`Current file: ${fileName}`);
        res.end();
    }
});
```

In this example, if the URL is /file-info, the server responds with the name of the current file.

Step 4: Use the Events Module

The events module allows you to create event-driven applications. Here's an example that demonstrates a custom event and an event listener.

```
const events = require('events');
const eventEmitter = new events.EventEmitter();

eventEmitter.on('customEvent', (message) => {
    console.log(`Custom event received: ${message}`);
});
```

```
server.on('request', (req, res) => {
    if (req.url === '/trigger-event') {
        eventEmitter.emit('customEvent', 'Hello from the custom event!');
        res.writeHead(200, { 'Content-Type': 'text/plain' });
        res.write('Custom event triggered.');
        res.end();
    }
});
```

This example defines a custom event and its handler. When the server receives a request with the URL /trigger-event, it triggers the custom event, sending a message to the console, and responds to the client with a confirmation message.

Conclusion

These examples demonstrate how to create a simple HTTP server using Node.js and explore modules like os, path, and events. Each module has unique capabilities that can be leveraged to build more complex Node.js applications.

EXPERIMENT: 10

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

To develop a simple Express.js web application that interacts with a REST API to perform CRUD (Create, Read, Update, Delete) operations on student data, you'll need to set up the Express.js framework, define the REST API endpoints, and create the necessary middleware to handle HTTP requests. Below is a step-by-step guide to creating such an application, with suggestions for testing with Postman.

Step 1: Set Up an Express.js Project

To start, create a new Node.js project and install the required dependencies:

```
# Initialize a new Node.js project
mkdir express-student-api
cd express-student-api
npm init -y

# Install Express.js
npm install express
```

Step 2: Define Student Data and CRUD Endpoints

Create a simple Express.js server that allows you to perform CRUD operations on student data. For this example, I'll use a basic in-memory array to store student information.

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

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

// In-memory student data
let students = [
  { id: 1, name: 'John Doe', age: 20 },
  { id: 2, name: 'Jane Smith', age: 22 },
];

// CREATE: Add a new student
```

```
app.post('/students', (req, res) => {
  const { name, age } = req.body;
  const newStudent = {
    id: students.length + 1, // Simple ID generation
    name,
    age,
  };
  students.push(newStudent);
  res.status(201).json(newStudent);
});

// READ: Get all students
app.get('/students', (req, res) => {
res.status(200).json(students);
});

// READ: Get a student by ID
app.get('/students/:id', (req, res) => {
const { id } = req.params;
  const student = students.find((s) => s.id === parseInt(id));
  if (student) {
    res.status(200).json(student);
  } else {
    res.status(404).json({ message: 'Student not found' });
  }
});

// UPDATE: Update a student's information by ID
app.put('/students/:id', (req, res) => {
  const { id } = req.params;
  const { name, age } = req.body;
  const student = students.find((s) => s.id === parseInt(id));

  if (student) {
    student.name = name || student.name;
    student.age = age || student.age;
    res.status(200).json(student);
  } else {
    res.status(404).json({ message: 'Student not found' });
  }
```

```
});

// DELETE: Delete a student by ID
app.delete('/students/:id', (req, res) => {
const { id } = req.params;
  const studentIndex = students.findIndex((s) => s.id === parseInt(id));

  if (studentIndex !== -1) {
    const deletedStudent = students.splice(studentIndex, 1)[0];
    res.status(200).json(deletedStudent);
  } else {
    res.status(404).json({ message: 'Student not found' });
  }
});

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

Step 3: Test the REST API with Postman

Now that you have a simple Express.js REST API, you can test CRUD operations using Postman or any other API testing tool.

CREATE:
Method: POST
URL: http://localhost:3000/students
Body: { "name": "New Student", "age": 21 }
Expected Response: Status 201 and the new student data
READ (all students):
Method: GET
URL: http://localhost:3000/students
Expected Response: Status 200 and the list of all students
READ (specific student):
Method: GET
URL: http://localhost:3000/students/1
Expected Response: Status 200 and the data for student with ID 1
UPDATE:
Method: PUT
URL: http://localhost:3000/students/1
Body: { "name": "Updated Name", "age": 23 }

Expected Response: Status 200 and the updated student data
DELETE:
Method: DELETE
URL: http://localhost:3000/students/1
Expected Response: Status 200 and the deleted student data

Conclusion

This guide provides a basic Express.js application with REST API endpoints for CRUD operations on student data. You can extend this application by adding a database backend, more complex validation, authentication, or additional features as needed. Use Postman or similar tools to test and validate the endpoints.

EXPERIMENT: 11

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

To create authorized endpoints using JSON Web Token (JWT), you'll need to add user authentication and authorization functionality to your Express.js application. This involves several steps: creating user login endpoints, generating JWT tokens, verifying tokens on protected routes, and handling token expiration.

Step 1: Install Required Packages

First, install the necessary packages for JWT-based authentication:

npm install jsonwebtoken bcryptjs

jsonwebtoken: A package to generate and verify JWT tokens.

bcryptjs: A package to hash and compare passwords for secure user authentication.

Step 2: Set Up User Authentication and JWT Generation

In this step, create an endpoint for user login, where a JWT token is generated upon successful authentication.

```
const express = require('express');
const jwt = require('jsonwebtoken');
const bcrypt = require('bcryptjs');
const app = express();
const PORT = 3000;

// Secret key for JWT signing
const JWT_SECRET = 'mySecretKey';

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

// Sample user data with hashed passwords (in a real application, this would be stored in a database)
const users = [
  {
    id: 1,
    username: 'admin',
    password: bcrypt.hashSync('adminpass', 10), // Hashed password
  },
```

```
];

// User login endpoint
app.post('/login', (req, res) => {
  const { username, password } = req.body;
  const user = users.find((u) => u.username === username);

  if (!user) {
    return res.status(401).json({ message: 'Invalid username or password' });
  }

  const passwordValid = bcrypt.compareSync(password, user.password);

  if (!passwordValid) {
    return res.status(401).json({ message: 'Invalid username or password' });
  }

  // Create JWT token with a payload containing user information
  const token = jwt.sign({ id: user.id, username: user.username }, JWT_SECRET, { expiresIn: '1h' });

  res.status(200).json({ token });
});
```

In this example, the /login endpoint verifies the user credentials and returns a JWT token upon successful authentication.

Step 3: Middleware to Protect Endpoints

Next, create middleware to verify JWT tokens for protected endpoints.

```
// Middleware to authenticate using JWT
const authenticateJWT = (req, res, next) => {
  const token = req.headers.authorization && req.headers.authorization.split(' ')[1];

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

  try {
    const decoded = jwt.verify(token, JWT_SECRET);
```

```
    req.user = decoded; // Attach decoded user info to the request object
    next();
  } catch (err) {
    return res.status(401).json({ message: 'Invalid or expired token' });
  }
};
```

This middleware checks for a JWT token in the Authorization header and verifies it. If the token is valid, the user information is attached to the request object; otherwise, it responds with an unauthorized message.

Step 4: Apply JWT Authentication to Protected Endpoints

Now, you can apply the JWT authentication middleware to specific endpoints to protect them.

```
// In-memory student data
let students = [
  { id: 1, name: 'John Doe', age: 20 },
  { id: 2, name: 'Jane Smith', age: 22 },
];

// CREATE: Add a new student (requires authentication)
app.post('/students', authenticateJWT, (req, res) => {
const { name, age } = req.body;
  const newStudent = {
    id: students.length + 1, // Simple ID generation
    name,
    age,
  };
  students.push(newStudent);
  res.status(201).json(newStudent);
});

// READ: Get all students (requires authentication)
app.get('/students', authenticateJWT, (req, res) => {
res.status(200).json(students);
});

// READ: Get a student by ID (requires authentication)
app.get('/students/:id', authenticateJWT, (req, res) => {
const { id } = req.params;
```

```
  const student = students.find((s) => s.id === parseInt(id));

  if (student) {
    res.status(200).json(student);
  } else {
    res.status(404).json({ message: 'Student not found' });
  }
});

// UPDATE: Update a student's information by ID (requires authentication)
app.put('/students/:id', authenticateJWT, (req, res) => {
  const { id } = req.params;
  const { name, age } = req.body;
  const student = students.find((s) => s.id === parseInt(id));

  if (student) {
    student.name = name || student.name;
    student.age = age or student.age;
    res.status(200).json(student);
  } else {
    return res.status(404).json({ message: 'Student not found' });
  }
});

// DELETE: Delete a student by ID (requires authentication)
app.delete('/students/:id', authenticateJWT, (req, res) => {
const { id } = req.params;
  const studentIndex = students.findIndex((s) => s.id === parseInt(id));

  if (studentIndex !== -1) {
    const deletedStudent = students.splice(studentIndex, 1)[0];
    res.status(200).json(deletedStudent);
  } else {
    res.status(404).json({ message: 'Student not found' });
  }
});

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

Step 5: Test the REST API with JWT Protection in Postman

To test the endpoints with JWT-based authentication, use Postman:

Login Endpoint:
Method: POST
URL: http://localhost:3000/login
Body: { "username": "admin", "password": "adminpass" }
Expected Response: JWT token in JSON format { "token": "..." }
Protected Endpoints:
Add an Authorization header with the format Bearer <token>, where <token> is the JWT obtained from the /login endpoint.
Test the endpoints /students, /students/:id, /students, /students/:id, /students/:id with the appropriate HTTP methods (GET, POST, PUT, DELETE).
Ensure that you use the JWT token for each request to the protected endpoints. If a valid token is not provided, the server should respond with a 401 Unauthorized status.

Conclusion

This guide shows how to implement JWT-based authentication and authorization for an Express.js web application with CRUD operations on student data. The use of middleware to verify JWT tokens allows you to secure specific endpoints and ensure that only authorized users can access them. Test your application with Postman or other similar tools to validate the JWT-based security.

EXPERIMENT: 12

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

To create a React application for a student management system with registration, login, contact, and about pages, you can follow a step-by-step approach to set up a basic React project, implement routing, and create the required pages. Here's a complete guide to achieve this.

Step 1: Set Up a React Project

First, set up a new React project using Create React App (CRA).

```
# Create a new React project
npx create-react-app student-management

# Change directory to the newly created project
cd student-management
```

Step 2: Install React Router for Navigation

To implement routing, you'll need React Router, which allows you to define navigation and manage page transitions in a React application.

```
# Install React Router
npm install react-router-dom
```

Step 3: Set Up Routing in the App Component

Create a basic routing structure in the main App component to manage navigation between pages.

Jsx

```
// src/App.js
import React from 'react';
import { BrowserRouter as Router, Route, Switch } from 'react-router-dom';
import Navbar from './components/Navbar';
import Home from './pages/Home';
import Registration from './pages/Registration';
import Login from './pages/Login';
import Contact from './pages/Contact';
import About from './pages/About';
```

```
function App() {
  return (
    <Router>
      <div className="App">
        <Navbar />
        <Switch>
          <Route exact path="/" component={Home} />
          <Route path="/register" component={Registration} />
          <Route path="/login" component={Login} />
          <Route path="/contact" component={Contact} />
          <Route path="/about" component={About} />
        </Switch>
      </div>
    </Router>
  );
}

export default App;
```

In this example, a Navbar component is used for navigation, and various routes are defined to handle different paths (/, /register, /login, /contact, and /about).

Step 4: Create the Navigation Bar

Create a simple navigation bar to allow users to navigate between different pages.

jsx
Copy code

```
// src/components/Navbar.js
import React from 'react';
import { Link } from 'react-router-dom';

function Navbar() {
  return (
    <nav>
      <ul>
        <li><Link to="/">Home</Link></li>
        <li><Link to="/register">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>
  );
}

export default Navbar;
```

This navigation bar allows users to navigate between pages using Link components from React Router.

Step 5: Create Page Components

Now, create the components for each of the pages: Home, Registration, Login, Contact, and About.

Home Page

jsx

Copy code

```
// src/pages/Home.js
import React from 'react';

function Home() {
  return (
    <div>
      <h1>Welcome to the Student Management System</h1>
      <p>Navigate through the registration, login, contact, or about pages using the navigation bar.</p>
    </div>
  );
}

export default Home;
```

Registration Page

jsx

Copy code

```
// src/pages/Registration.js
import React, { useState } from 'react';

function Registration() {
  const [formData, setFormData] = useState({
    username: '',
    password: '',
    email: '',
```

```
  });

  const handleChange = (e) => {
    const { name, value } = e.target;
    setFormData({ ...formData, [name]: value });
  };

  const handleSubmit = (e) => {
    e.preventDefault();
    console.log('Registration data:', formData);
    // In a real application, you'd send this data to a server for processing
  };

  return (
    <div>
      <h1>Register</h1>
      <form onSubmit={handleSubmit}>
        <div>
          <label>Username:</label>
          <input type="text" name="username" value={formData.username}
onChange={handleChange} />
        </div>
        <div>
          <label>Password:</label>
          <input type="password" name="password" value={formData.password}
onChange={handleChange} />
        </div>
        <div>
          <label>Email:</label>
          <input type="email" name="email" value={formData.email}
onChange={handleChange} />
        </div>
        <button type="submit">Register</button>
      </form>
    </div>
  );
}

export default Registration;
```

Login Page

jsx
Copy code

```
// src/pages/Login.js
import React, { useState } from 'react';

function Login() {
  const [formData, setFormData] = useState({
    username: '',
    password: '',
  });

  const handleChange = (e) => {
    const { name, value } = e.target;
    setFormData({ ...formData, [name]: value });
  };

  const handleSubmit = (e) => {
    e.preventDefault();
    console.log('Login data:', formData);
    // In a real application, you'd authenticate the user with a server
  };

  return (
    <div>
      <h1>Login</h1>
      <form onSubmit={handleSubmit}>
        <div>
          <label>Username:</label>
          <input type="text" name="username" value={formData.username}
onChange={handleChange} />
        </div>
        <div>
          <label>Password:</label>
          <input type="password" name="password" value={formData.password}
onChange={handleChange} />
        </div>
        <button type="submit">Login</button>
      </form>
    </div>
  );
```

```
}

export default Login;
```
Contact Page
jsx
Copy code
```
// src/pages/Contact.js
import React from 'react';

function Contact() {
  return (
    <div>
      <h1>Contact Us</h1>
      <p>Email: contact@school.com</p>
      <p>Phone: 123-456-7890</p>
    </div>
  );
}

export default Contact;
```
About Page
jsx
Copy code
```
// src/pages/About.js
import React from 'react';

function About() {
  return (
    <div>
      <h1>About the Student Management System</h1>
      <p>This system is designed to manage student information and facilitate registration,
login, and other functionalities.</p>
    </div>
  );
}

export default About;
```
Conclusion

This guide provides a basic React application with a navigation bar and routing to create a student management system with registration, login, contact, and about pages. Each

page has a simple structure, and routing is implemented using React Router. To run the application, use the following command:

bash
Copy code
npm start
This command starts the development server, allowing you to navigate through the application and test the routing functionality. To enhance this application, consider adding state management, user authentication, and interactions with a backend server for storing and retrieving data.

EXPERIMENT: 13

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

To create a React service that fetches weather information from OpenWeatherMap and displays current and historical weather data using Chart.js, you need to follow these steps:

Set Up a React Project
Install Required Dependencies
Create a Weather Service to Fetch Data
Display Weather Information
Visualize Weather Data with Chart.js
I'll walk you through each step in detail, including setting up the React project, creating a service to fetch weather information from OpenWeatherMap, and using Chart.js to represent data graphically.

Step 1: Set Up a React Project
First, create a new React project using Create React App.

bash
Copy code
npx create-react-app weather-app
cd weather-app

Step 2: Install Required Dependencies
Install the packages needed for interacting with OpenWeatherMap and creating charts with Chart.js.

bash
Copy code
Install Axios for making HTTP requests
npm install axios

Install Chart.js and React-Chartjs-2 for graphical representation
npm install chart.js react-chartjs-2

Step 3: Create a Weather Service to Fetch Data
OpenWeatherMap provides weather data via an API. You need to sign up for an API key and use it to fetch weather data.

Create a service to fetch weather data:

jsx
Copy code

```
// src/services/weatherService.js
import axios from 'axios';

const API_KEY = 'YOUR_OPENWEATHERMAP_API_KEY'; // Replace with your API key
const BASE_URL = 'https://api.openweathermap.org/data/2.5';

const getWeather = (city) => {
  return axios.get(`${BASE_URL}/weather`, {
    params: {
      q: city,
      appid: API_KEY,
      units: 'metric', // Use metric system for temperatures in Celsius
    },
  });
};

const getHistoricalWeather = (lat, lon) => {
  const currentTimestamp = Math.floor(Date.now() / 1000);
  const oneDaySeconds = 86400;

  return axios.get(`${BASE_URL}/timemachine`, {
    params: {
      lat,
      lon,
      dt: currentTimestamp - (7 * oneDaySeconds), // One week ago
      appid: API_KEY,
      units: 'metric',
    },
  });
};

export { getWeather, getHistoricalWeather };
```

This service provides two functions:

getWeather: Fetches the current weather for a given city.
getHistoricalWeather: Fetches historical weather data for a given latitude and longitude.

Step 4: Display Weather Information

Create a React component that fetches and displays the current weather for a specified city.

jsx
Copy code

```
// src/components/WeatherDisplay.js
import React, { useEffect, useState } from 'react';
import { getWeather } from '../services/weatherService';

function WeatherDisplay({ city }) {
  const [weather, setWeather] = useState(null);

  useEffect(() => {
    const fetchWeather = async () => {
      try {
        const response = await getWeather(city);
        setWeather(response.data);
      } catch (error) {
        console.error('Error fetching weather data:', error);
      }
    };

    fetchWeather();
  }, [city]);

  if (!weather) {
    return <div>Loading...</div>;
  }

  const { main, weather: weatherDetails, name } = weather;

  return (
    <div>
      <h2>Weather in {name}</h2>
      <p>Temperature: {main.temp} °C</p>
      <p>Weather: {weatherDetails[0].description}</p>
    </div>
```

```
  );
}

export default WeatherDisplay;
```

This component fetches the current weather for a specified city and displays it.

Step 5: Visualize Weather Data with Chart.js

Create a React component that fetches historical weather data and displays it using Chart.js.

jsx

Copy code

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

function WeatherChart({ city }) {
  const [chartData, setChartData] = useState(null);

  useEffect(() => {
    const fetchHistoricalWeather = async () => {
      try {
        const currentWeather = await getWeather(city);
        const { coord } = currentWeather.data;

        const historicalWeather = await getHistoricalWeather(coord.lat, coord.lon);

        const temperatureData = historicalWeather.data.hourly.map((hour) => ({
          x: new Date(hour.dt * 1000),
          y: hour.temp,
        }));

        setChartData({
          datasets: [
            {
              label: 'Temperature over Time',
              data: temperatureData,
              borderColor: 'rgba(75, 192, 192, 1)',
              borderWidth: 2,
```

```
            fill: false,
          },
        ],
      });
    } catch (error) {
      console.error('Error fetching historical weather:', error);
    }
  };

  fetchHistoricalWeather();
 }, [city]);

 if (!chartData) {
  return <div>Loading...</div>;
 }

 return (
  <div>
   <h2>Temperature Chart for {city}</h2>
   <Line data={chartData} />
  </div>
 );
}

export default WeatherChart;
```

This component fetches historical weather data for the specified city and displays it using a line chart from Chart.js.

Step 6: Integrate Components in the App

Now, integrate the WeatherDisplay and WeatherChart components into the main App component.

jsx
Copy code

```
// src/App.js
import React from 'react';
import WeatherDisplay from './components/WeatherDisplay';
import WeatherChart from './components/WeatherChart';

function App() {
```

```
  const city = 'New York'; // Example city name

  return (
    <div className="App">
      <h1>Weather Information</h1>
      <WeatherDisplay city={city} />
      <WeatherChart city={city} />
    </div>
  );
}

export default App;
```

This App component demonstrates how to use both components to display current weather information and a temperature chart.

Conclusion

With this guide, you have created a React service to fetch weather information from OpenWeatherMap and display it using Chart.js for graphical representation. The WeatherDisplay component fetches and displays current weather, while the WeatherChart component visualizes historical weather data. You can extend this application with additional features like user input for city names, caching data to reduce API calls, or more complex graphical representations. Remember to replace 'YOUR_OPENWEATHERMAP_API_KEY' with your actual OpenWeatherMap API key.

EXPERIMENT: 14

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

Creating a TODO application in React involves several key steps: setting up the React project, creating necessary components for the TODO functionality, adding CRUD (Create, Read, Update, Delete) operations, and deploying the application to GitHub Pages. This guide will walk you through each step, from project setup to deployment.

Step 1: Set Up the React Project

Start by creating a new React project using Create React App.

bash

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

Step 2: Create TODO Components

Create components to manage the TODO list, including adding new tasks, listing existing tasks, updating tasks, and deleting them. Here's a basic structure for the components.

Task Component

A component that represents a single TODO item with the ability to mark it as complete or delete it.

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

function Task({ task, onToggle, onDelete }) {
  return (
    <div>
      <input
        type="checkbox"
        checked={task.completed}
        onChange={() => onToggle(task.id)}
```

```
      />
      <span style={{ textDecoration: task.completed ? 'line-through' : 'none' }}>
       {task.text}
      </span>
      <button onClick={() => onDelete(task.id)}>Delete</button>
    </div>
  );
}

export default Task;
```

Task List Component

A component that displays a list of TODO items and allows for toggling completion and deletion.

jsx

Copy code

```
// // src/components/TaskList.js

import React from 'react';
import Task from './Task';

function TaskList({ tasks, onToggle, onDelete }) {
  return (
    <div className="task-list">
      {tasks.map((task) => (
        <Task
          key={task.id}
          task={task}
          onToggle={onToggle}
          onDelete={onDelete}
        />
      ))}
    </div>
  );
}export default TaskList;
```

Add Task Component

A component that provides a form to add a new task to the TODO list.

```
// src/components/AddTask.js

import React, { useState } from 'react';

function AddTask({ onAdd }) {
  const [taskText, setTaskText] = useState('');

  const handleSubmit = (e) => {
    e.preventDefault();
    if (taskText.trim() === '') {
      return;
    }
    onAdd(taskText);
    setTaskText('');
  };

  return (
    <form onSubmit={handleSubmit} style={styles.form}>
      <input
        type="text"
        placeholder="Enter a new task"
        value={taskText}
        onChange={(e) => setTaskText(e.target.value)}
        style={styles.input}
      />
      <button type="submit" style={styles.button}>
        Add Task
      </button>
    </form>
  );
}

const styles = {
  form: {
    display: 'flex',
    gap: '10px',
    marginBottom: '20px',
  },
  input: {
    flexGrow: 1,
    padding: '8px',
    fontSize: '16px',
  },
  button: {
    padding: '8px 16px',
    fontSize: '16px',
```

```
    backgroundColor: '#3498db',
    color: '#fff',
    border: 'none',
    borderRadius: '4px',
    cursor: 'pointer',
  },
};

export default AddTask;
```

Step 3: Create the App Component

Now, create the **`App`** component to manage the overall state of the TODO list, handle task additions, toggles, and deletions.

jsx

```
// src/App.js

import React, { useState } from 'react';
import TaskList from './components/TaskList';
import AddTask from './components/AddTask';

function App() {
  const [tasks, setTasks] = useState([]);

  const addTask = (text) => {
    const newTask = {
      id: tasks.length + 1,
      text,
      completed: false,
    };
    setTasks([...tasks, newTask]);
  };

  const toggleTask = (id) => {
    setTasks(
      tasks.map((task) =>
        task.id === id ? { ...task, completed: !task.completed } : task
      )
    );
  };

  const deleteTask = (id) => {
    setTasks(tasks.filter((task) => task.id !== id));
  };

  return (
    <div className="App" style={styles.container}>
      <h1 style={styles.heading}>TODO Application</h1>
      <AddTask onAdd={addTask} />
      <TaskList tasks={tasks} onToggle={toggleTask} onDelete={deleteTask} />
    </div>
  );
}
```

```
const styles = {
  container: {
    maxWidth: '600px',
    margin: '40px auto',
    padding: '20px',
    borderRadius: '8px',
    backgroundColor: '#f8f9fa',
    boxShadow: '0 2px 8px rgba(0,0,0,0.1)',
  },
  heading: {
    textAlign: 'center',
    color: '#2c3e50',
    marginBottom: '20px',
  },
};

export default App;
```

This **App**component provides a simple TODO list application with task creation, completion toggling, and deletion functionality.

Step 4: Deploy the Application to GitHub Pages

To deploy your application to GitHub Pages, you need to set up the necessary configurations.

Install gh-pages Package

Install the `gh-pages` package to manage the deployment process.

```bash
npm install gh-pages --save-dev
```

Configure `package.json`

Update **package.json**to set up the deployment script and specify the GitHub repository for deployment.

```json
```

```
{
  "name": "todo-app",
  "version": "0.1.0",
  "private": true,
  "dependencies": {
    "react": "^17.0.2",
    "react-dom": "^17.0.2",
    "react-scripts": "4.0.3"
  },
  "scripts": {
    "start": "react-scripts start",
    "build": "react-scripts build",
    "test": "react-scripts test",
    "eject": "react-scripts eject",
    "predeploy": "npm run build",
    "deploy": "gh-pages -d build"
  },
  "homepage": "https://<your-github-username>.github.io/todo-app"
}
```

Replace **`<your-github-username>`** with your actual GitHub username.

Push the Project to GitHub

Create a new GitHub repository, initialize your local Git repository, and push the project to GitHub.

bash

```
# Initialize Git repository
git init
 # Add and commit files
git add .
git commit -m "Initial commit for TODO app"
# Add GitHub remote repository
 git remote add origin https://github.com/<your-github-username>/todo-app.git
# Push to GitHub
git push -u origin main
```

Replace **`<your-github-username>`** with your GitHub username.

Deploy to GitHub Pages

Finally, deploy the application to GitHub Pages.

bash

```
npm run deploy
```

This command will build the application and deploy it to GitHub Pages. You can then visit the URL specified in **package.json** to see your deployed TODO application.

Conclusion

Following this guide, you have created a simple TODO application in React with task creation, completion, and deletion functionality. You also deployed the application to GitHub Pages, allowing you to share it with others or use it as a live demonstration. This is a foundational project that you can expand with additional features like persistence, user authentication, or more complex task management.