Contents

1 Cycle 1 Program 1 : Responsive Personal Portfolio Website

Problem Statement

Create a responsive personal portfolio website using HTML5 and CSS3. The website should include sections for your profile, skills, experience, projects, and contact information. Use Tailwind CSS to style the website and ensure that it is responsive on different screen sizes.

Expected Output

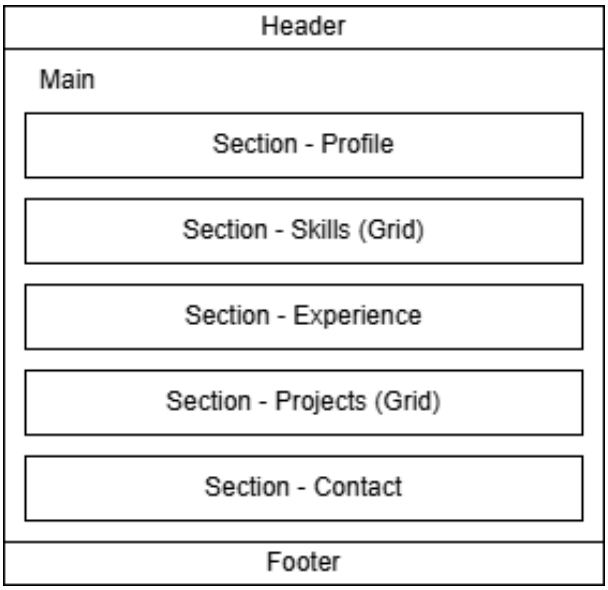

Figure 1: Responsive Personal Portfolio Website: Expected Output

Code

index.html

```
<html>
    <head>
        <meta name="viewport" content="width=device-width,
           initial-scale=1.0">
        <title>Personal Portfolio</title>
    <script src="https://cdn.tailwindcss.com/"></script>
    </head>
    <body class="bg-green-200">
        <header class="bg-blue-500 text-white p-4">
            <h1 class="text-2xl text-center">Personal Portfolio</h1>
        </header>
        <main class="container mx-auto p-4">
            <!-- Profile section -->
             <section id="profile" class="mt-4">
                <h2 class="text-xl font-bold mb-4">Profile</h2>
```

```
15                  <div class="bg-white p-4 rounded-lg">
16                  <p>I am a student at the SNIST. I love learning new
                        things.</p>
17                  </div>
18              </section>
19              <!-- Skills section -->
20              <section id="skills">
21                  <h2 class="text-xl font-bold mb-4">Skills</h2>
22                  <div class="bg-white p-4 rounded-lg grid gap-4
                        sm:grid-cols-2 md:grid-cols-4 lg:grid-cols-6">
23                      <div class="bg-pink-500 text-white p-2
                            rounded-lg">HTML5</div>
24                      <div class="bg-pink-500 text-white p-2
                            rounded-lg">CSS3</div>
25                      <div class="bg-pink-500 text-white p-2
                            rounded-lg">JavaScript</div>
26                      <div class="bg-pink-500 text-white p-2
                            rounded-lg">Python</div>
27                      <div class="bg-pink-500 text-white p-2
                            rounded-lg">Java</div>
28                      <div class="bg-pink-500 text-white p-2
                            rounded-lg">C++</div>
29                      <div class="bg-pink-500 text-white p-2
                            rounded-lg">SQL</div>
30                      <div class="bg-pink-500 text-white p-2
                            rounded-lg">Git</div>
31                      <div class="bg-pink-500 text-white p-2
                            rounded-lg">Linux</div>
32                      <div class="bg-pink-500 text-white p-2
                            rounded-lg">Windows</div>
33                      <div class="bg-pink-500 text-white p-2
                            rounded-lg">React</div>
34                      <div class="bg-pink-500 text-white p-2
                            rounded-lg">Angular</div>
35                  </div>
36              </section>
37              <!-- Experience section -->
38              <section id="experience">
39                  <h2 class="text-xl font-bold mb-4">Experience</h2>
40                  <div class="bg-white p-4 rounded-lg">
41                      <ul>
42                          <li>Internship at ABC Company</li>
43                          <li>Internship at XYZ Company</li>
44                      </ul>
45                  </div>
46              </section>
47              <!-- Projects section -->
48              <section id="projects">
49                  <h2 class="text-xl font-bold mb-4">Projects</h2>
50                  <div class="bg-white p-4 rounded-lg">
51                      <div>
52                          <h3>Project 1</h3>
53                          <p>Description of project 1</h3>
54                      </div>
55                      <div>
56                          <h3>Project 2</h3>
57                          <p>Description of project 2</h3>
58                      </div>
```

```
            </div>
        </section>
        <!-- Contact section -->
        <section id="contact">
            <h2 class="text-xl font-bold mb-4">Contact</h2>
            <div class="bg-white p-4 rounded-lg">
                <p>Email: csea@snist.com</p>
                <p>Phone: 123-456-7890</p>
            </div>
        </section>
    </main>
    <footer class="bg-blue-500 text-white p-4">
        <p>&copy; 2025 Personal Portfolio</p>
    </footer>
</body>
</html>
```

Output Screens

Figure 2: Responsive Personal Portfolio Website: Output with Medium screen size

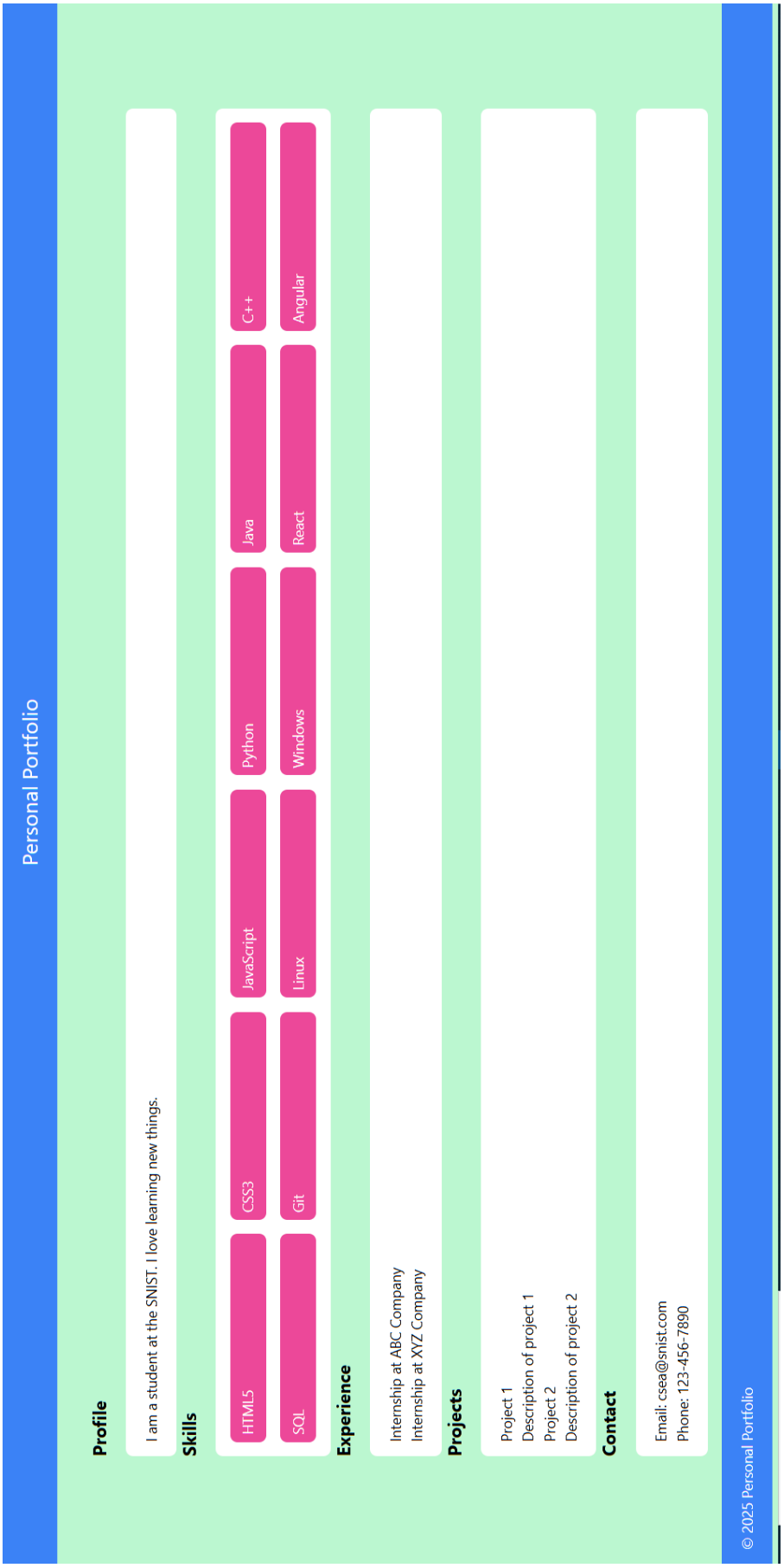

Figure 3: Responsive Personal Portfolio Website: Output with Large screen size

2 Cycle 1 Program 2 : E-commerce Product Page

Problem Statement

Design and develop an e-commerce product page using HTML5, CSS3, and Tailwind CSS.The page should display product information, including images, descriptions,prices, and options for adding to cart. Use Tailwind CSS to create a visually appealing and consistent layout.

Expected Output

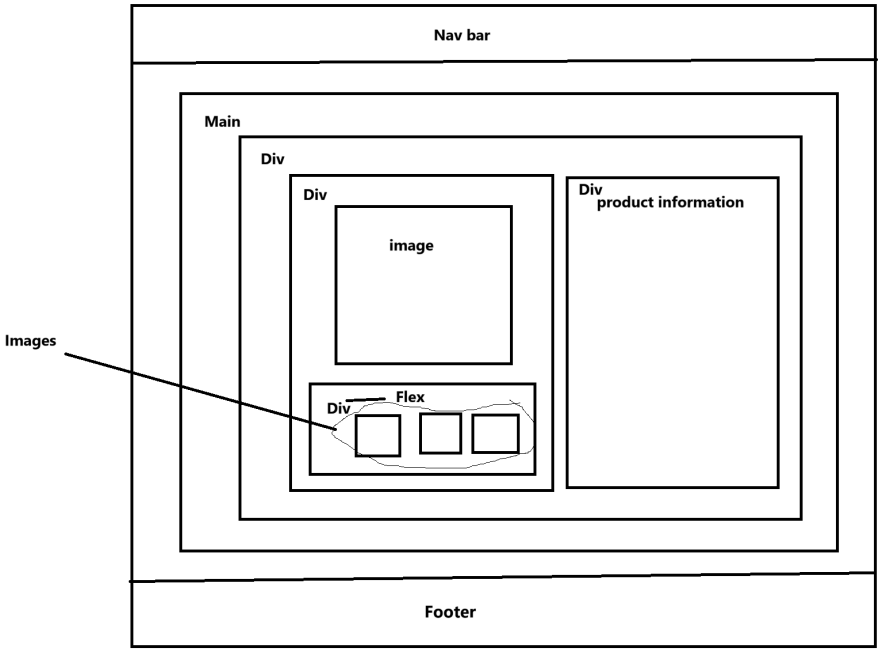

Figure 4: E-commerce Product Page : Expected Output

Code

index.html

```
1
2  <html>
3      <head>
4          <!-- <script src="https://cdn.tailwindcss.com"></script> -->
5          <link href="./src/output.css" rel="stylesheet">
6          <title>E-Shop</title>
7      </head>
8      <body class="bg-gradient-to-r from-blue-500 to-red-500 p-8" >
9          <nav class="bg-blue-900 text-white p-4 rounded-lg">
10             <div class="flex justify-between flex-row">
11                 <a href="#" class="font-bold mb-4">E-Shop</a>
12                 <ul class="flex space-x-6">
13                     <li><a href="#"
                           class="hover:text-blue-500">Home</a></li>
14                     <li><a href="#"
                           class="hover:text-blue-500">Products</a></li>
15                     <li><a href="#"
                           class="hover:text-blue-500">About</a></li>
16                     <li><a href="#"
                           class="hover:text-blue-500">Contact</a></li>
17                 </ul>
18             </div>
19         </nav>
20         <main class="container mx-auto my-8 p-4 bg-white text-black
               rounded-lg">
21         <!-- Product Section -->
22         <div class="flex md:space-x-8">
23         <!-- Product images -->
24         <div class="w-1/2">
25         <img src="./images/2.jpg" alt="Main Image" class="w-full
               shadow-xl h-auto rounded-lg">
26         <div class="flex space-x-6 mt-4">
27         <img src="./images/1.jpg" alt="Thumbnail_1" class="rounded-lg
               shadow-l w-1/3 h-auto transition-transform duration-300
               hover:scale-105">
28         <img src="./images/3.jpg" alt="Thumbnail_2" class="rounded-lg
               w-1/3 shadow-l h-auto transition-transform duration-300
               hover:scale-105">
29         <img src="./images/4.jpg" alt="Thumbnail_3" class="rounded-lg
               w-1/3 shadow-l h-auto transition-transform duration-300
               hover:scale-105">
30         </div>
31         </div>
32         <!-- Product info -->
33         <div class="w-1/2 mt-8">
34         <h1 class="text-2xl font-bold mb-4">ASUS ROG Strix Gaming
               Laptop</h1>
35         <p class="text-sm ">
36         Elevate your gaming with the ASUS ROG Strix Gaming Laptop,
               designed for peak performance and immersive visuals.
               Powered by Intel Core i7/i9 or AMD Ryzen processors and
               NVIDIA GeForce RTX 30-Series graphics, this laptop ensures
               smooth gameplay and fast rendering.
37         <h3 class="text-md font-bold mb-2">Key Features:</h3>
```

```
        <ul class="text-sm ">
        <li>144Hz/240Hz Display with 3ms response time for
            ultra-responsive gaming.</li>
        <li>ROG Intelligent Cooling system for optimal
            performance.</li>
        <li>Dolby Atmos Audio for rich, dynamic sound.</li>
        <li>Aura Sync RGB lighting for a personalized look.</li>
        <li>Comprehensive Connectivity with Wi-Fi 6, USB Type-C, and
            HDMI 2.1.</li>
        <li>Up to 64GB RAM and 1TB SSD for fast storage and
            multitasking.</li>
        <li>Perfect for gamers and power users seeking a
            high-performance laptop.</li>
        </ul>
        </p>
        <div class="flex items-center mb-4 mt-4">
        <label for="quantity" class="mr-5 text-sm font-bold
            mb-2">Quantity:</label>
        <input type="number" id="quantity" name="quantity" min="1"
            max="10" class="w-17 p-2 border border-gray-400 rounded-lg">
        </div>
        <button class="bg-red-500 text-white shadow-xl  p-4 rounded-lg
            transition-transform duration-300 hover:scale-105
            hover:bg-red-600 hover:text-white">Add to Cart</button>
        </div>
        </div>
        </main>
        <footer></footer>
    </body>
</html>
```

Output Screens

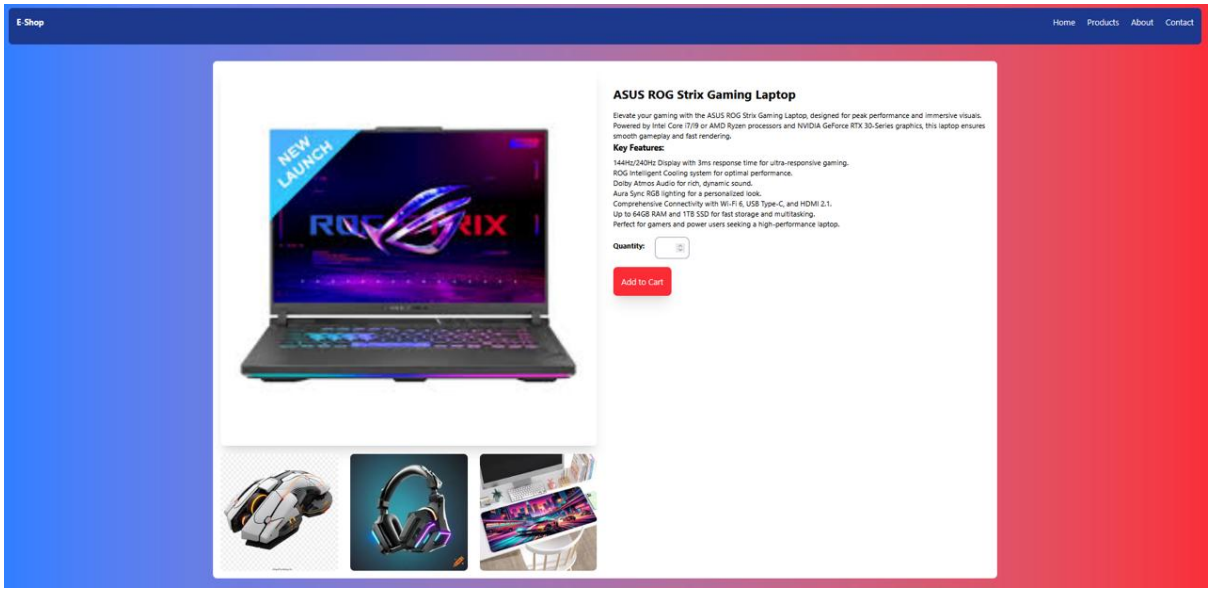

Figure 5: E-commerce Product Page : Output

3 Cycle 1 Program 3 : Interactive Blog Post with Comments

Problem Statement

Create an interactive blog post with comments using HTML5, CSS3, and JavaScript. The blog post should include a title, author, content, and a comment section. Use JavaScript to enable users to submit comments and display them on the page.

Expected Output

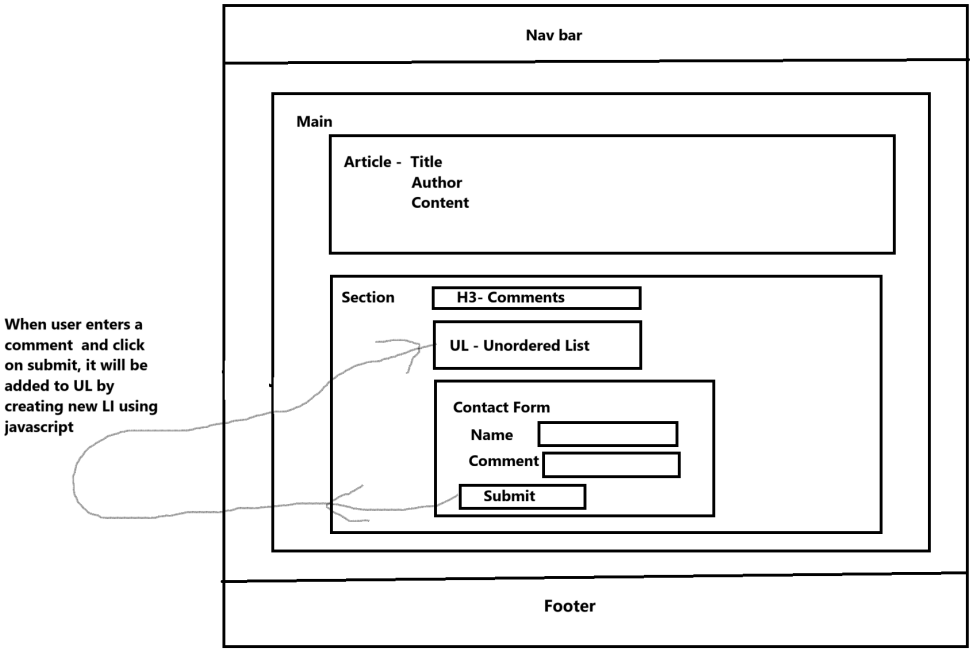

Figure 6: Interactive Blog Post with Comments: Expected Output

Code

index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width,
       initial-scale=1.0">
    <title>Interactive Blog Post</title>
    <style>
        body {
            font-family: Arial, sans-serif;
            margin: 20px;
            padding: 20px;
            background-color: #f4f4f4;
```

```
        }
        .blog-container {
            max-width: 600px;
            margin: auto;
            background: white;
            padding: 20px;
            border-radius: 10px;
            box-shadow: 0px 0px 10px rgba(0, 0, 0, 0.1);
        }
        .comment-section {
            margin-top: 20px;
        }
        .comment {
            background: #e9e9e9;
            padding: 10px;
            border-radius: 5px;
            margin-top: 10px;
        }
        input, button {
            margin-top: 10px;
            padding: 10px;
            width: 100%;
        }
    </style>
</head>
<body>
    <div class="blog-container">
        <h2>Understanding JavaScript Events</h2>
        <p><strong>Author:</strong> John Doe</p>
        <p>JavaScript events are actions that can be detected by your
           web application. Examples include clicks, key presses, and
           mouse movements...</p>

        <div class="comment-section">
            <h3>Comments</h3>
            <div id="comments"></div>
            <input type="text" id="commentInput" placeholder="Write a
               comment...">
            <button onclick="addComment()">Submit</button>
        </div>
    </div>
    <script>
        function addComment() {
            let commentText =
               document.getElementById("commentInput").value;
            if (commentText.trim() === "") return;

            let commentBox = document.createElement("div");
            commentBox.classList.add("comment");
            commentBox.textContent = commentText;

            document.getElementById("comments").appendChild(commentBox);
            document.getElementById("commentInput").value = "";
        }
    </script>
</body>
</html>
```

Output Screens

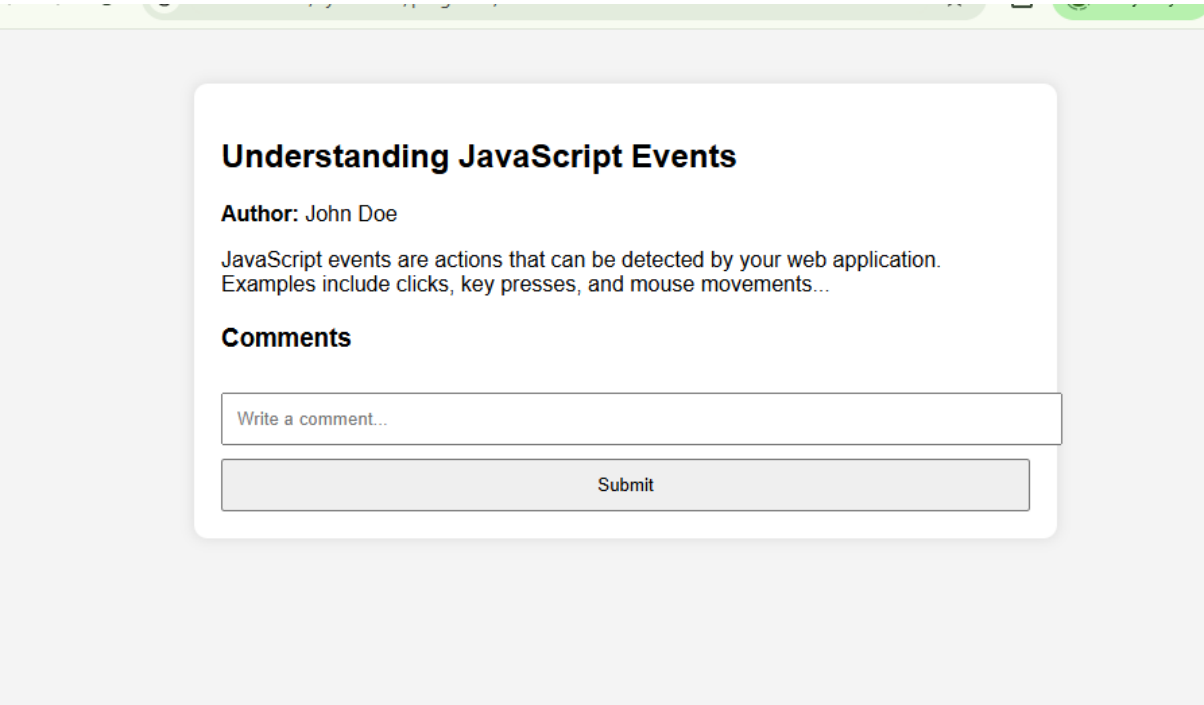

Figure 7: Interactive Blog Post with Comments: Output

4 Cycle 1 Program 4 : Adaptive Landing Page for Different Devices

Problem Statement

Develop an adaptive landing page that adjusts its layout and content based on the user's device. Use HTML5, CSS3, and JavaScript to detect the device type and display the appropriate content. Employ media queries and responsive design techniques to ensure the page looks great on all devices.

Expected Output

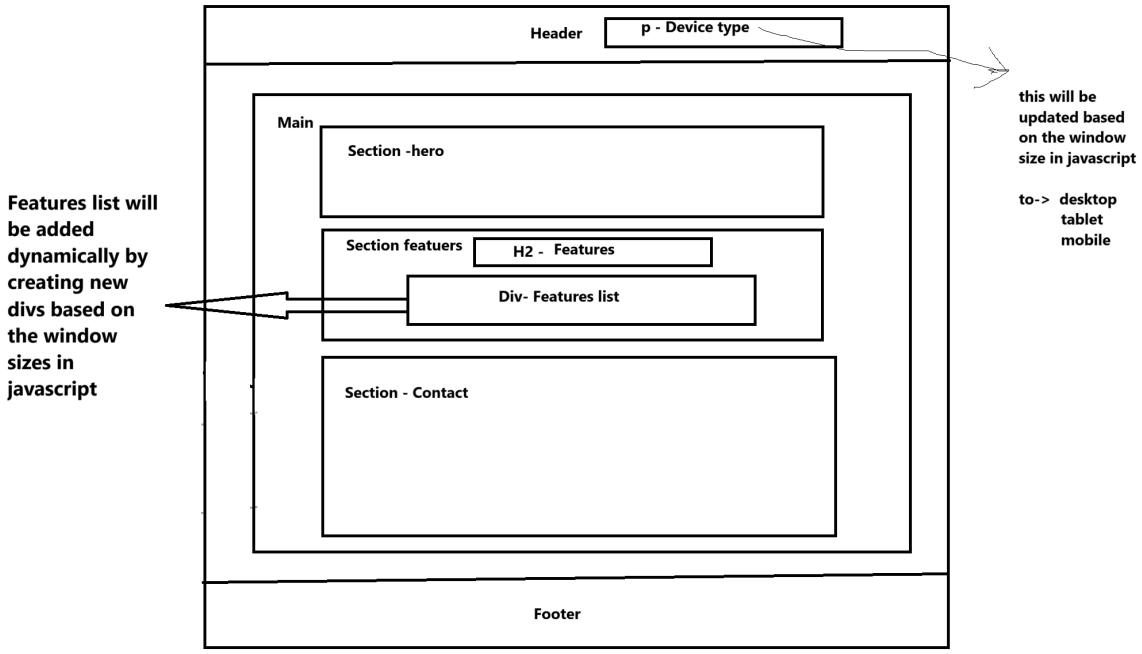

Figure 8: Adaptive Landing Page for Different Devices: Expected Output

Code

index.html

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

```
        <h1>Welcome to Our Service</h1>
        <p id="deviceType"></p>
    </header>
    <main>
        <section id="hero">
            <h2 id="heroTitle">Discover Our Solutions</h2>
            <p id="heroDescription">We provide top-notch solutions
               tailored to your needs.</p>
        </section>
        <section id="features">
            <h2>Features</h2>
            <div id="featureList">
                <!-- Features will be dynamically inserted here -->
            </div>
        </section>
        <section id="contact">
            <h2>Contact Us</h2>
            <p>Get in touch with us for more information.</p>
            <form id="contactForm">
                <label for="name">Name:</label>
                <input type="text" id="name" name="name" required>
                <label for="email">Email:</label>
                <input type="email" id="email" name="email" required>
                <label for="message">Message:</label>
                <textarea id="message" name="message" rows="4"
                   required></textarea>
                <button type="submit">Send Message</button>
            </form>
        </section>
    </main>
    <footer>
        <p>&copy; 2024 Our Service</p>
    </footer>
    <script src="script.js"></script>
</body>
</html>
```

script.js

```
// Function to detect device type
function detectDevice() {
    var deviceTypeElement = document.getElementById('deviceType');
    var featureList = document.getElementById('featureList');

    if (window.innerWidth < 768) {
        deviceTypeElement.textContent = "You are using a mobile
           device.";
        // Example mobile features
        featureList.innerHTML = `
            <div class="feature">Mobile Feature 1</div>
            <div class="feature">Mobile Feature 2</div>
        `;
    } else if (window.innerWidth < 1024) {
        deviceTypeElement.textContent = "You are using a tablet.";
        // Example tablet features
        featureList.innerHTML = `
            <div class="feature">Tablet Feature 1</div>
```

```
            <div class="feature">Tablet Feature 2</div>
            <div class="feature">Tablet Feature 3</div>
        `;
    } else {
        deviceTypeElement.textContent = "You are using a desktop.";
        // Example desktop features
        featureList.innerHTML = `
            <div class="feature">Desktop Feature 1</div>
            <div class="feature">Desktop Feature 2</div>
            <div class="feature">Desktop Feature 3</div>
            <div class="feature">Desktop Feature 4</div>
        `;
    }
}
// Detect device type on load and resize
window.onload = detectDevice;
window.onresize = detectDevice;
```

style.css

```
/* Basic reset */
* {
    margin: 0;
    padding: 0;
    box-sizing: border-box;
}
/* Styling for the header */
header {
    background-color: lightcoral;
    color: white;
    padding: 20px;
    text-align: center;
}
/* Styling for the main content */
main {
    padding: 20px;
    background-color: #f9f9f9;
}
/* Styling for the hero section */
#hero {
    text-align: center;
    margin-bottom: 20px;
}
#heroTitle {
    font-size: 2em;
    margin-bottom: 10px;
}
#heroDescription {
    font-size: 1.2em;
}
/* Styling for the features section */
#features {
    margin-bottom: 20px;
}
#featureList {
    display: flex;
    flex-wrap: wrap;
```

```
    gap: 10px;
}
.feature {
    background-color: white;
    padding: 20px;
    border: 1px solid #ddd;
    border-radius: 5px;
    flex: 1;
}
/* Styling for the contact form */
#contactForm {
    display: flex;
    flex-direction: column;
}
#contactForm label {
    margin-top: 10px;
}
#contactForm input,
#contactForm textarea {
    margin-top: 5px;
    padding: 10px;
    border: 1px solid #ddd;
    border-radius: 5px;
}
#contactForm button {
    margin-top: 10px;
    padding: 10px 20px;
    background-color: lightcoral;
    border: none;
    border-radius: 5px;
    color: white;
    cursor: pointer;
}
#contactForm button:hover {
    background-color: darkred;
}
/* Styling for the footer */
footer {
    text-align: center;
    padding: 10px;
    background-color: #333;
    color: white;
}
/* Media Queries for responsiveness */
/* For tablets and larger devices */
@media (min-width: 768px) {
    #featureList {
        flex-direction: row;
    }
}
/* For desktops and larger devices */
@media (min-width: 1024px) {
    #hero {
        text-align: left;
    }
    #heroTitle {
        font-size: 2.5em;
    }
```

```
96      #heroDescription {
97          font-size: 1.5em;
98      }
99  }
100
```

Output Screens

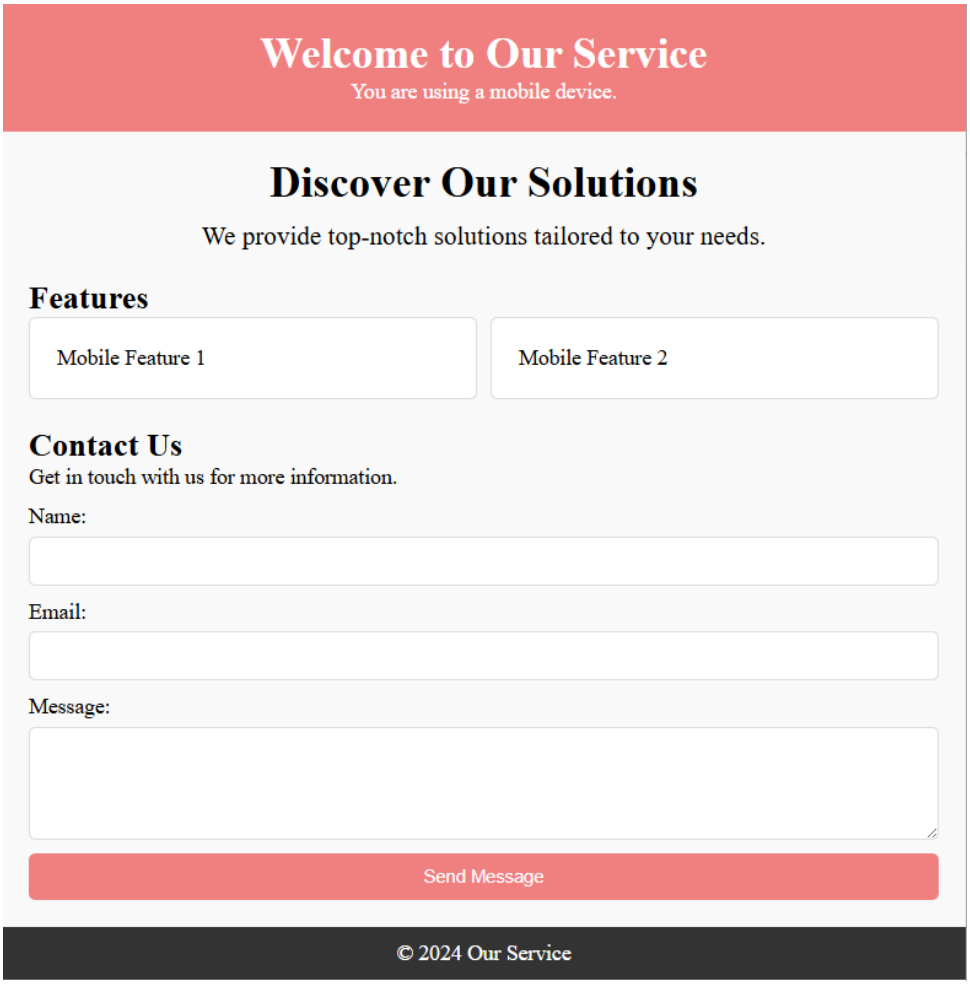

Figure 9: Adaptive Landing Page for Mobile Devices: Output

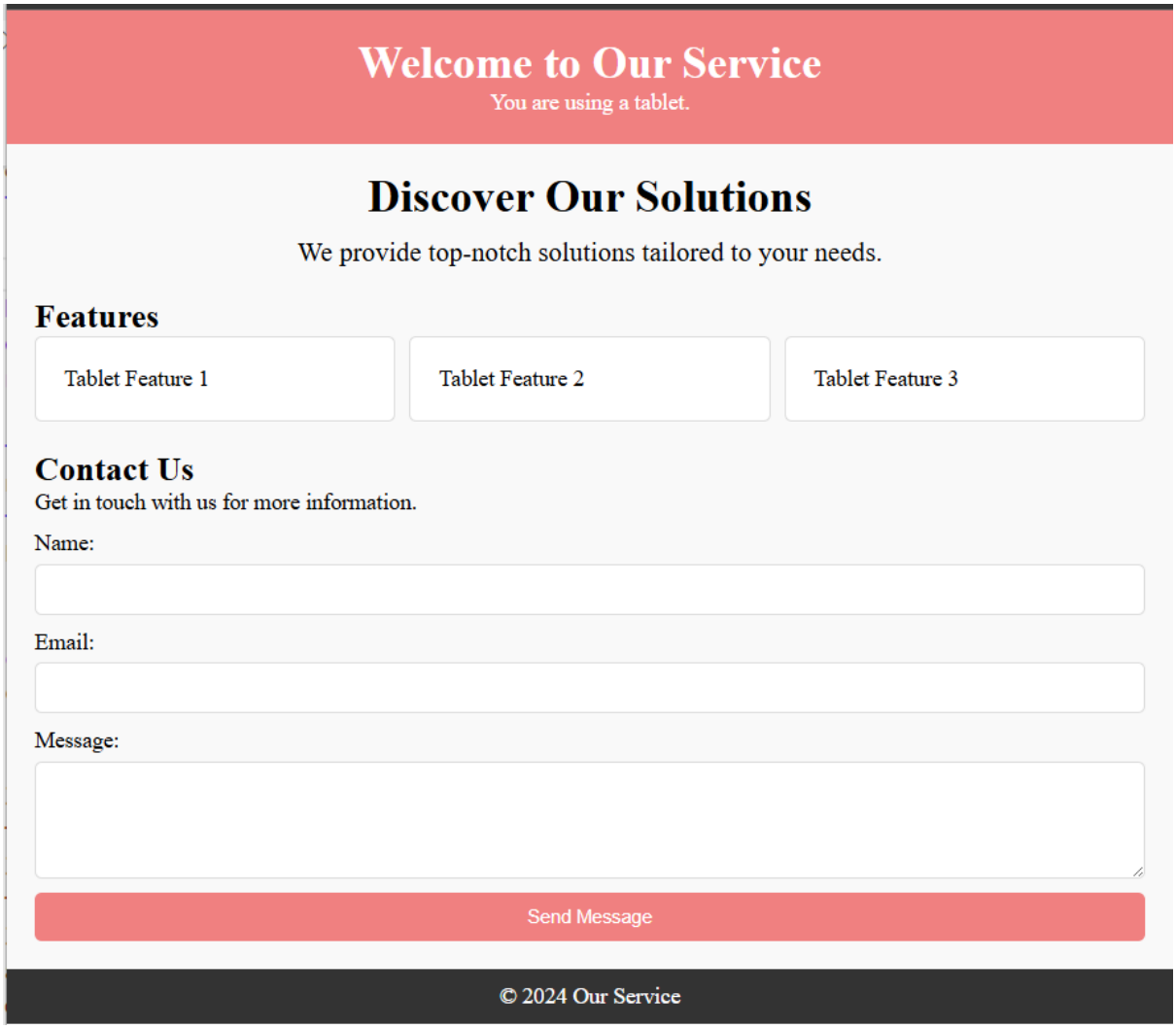

Figure 10: Adaptive Landing Page for Tablet Devices: Output

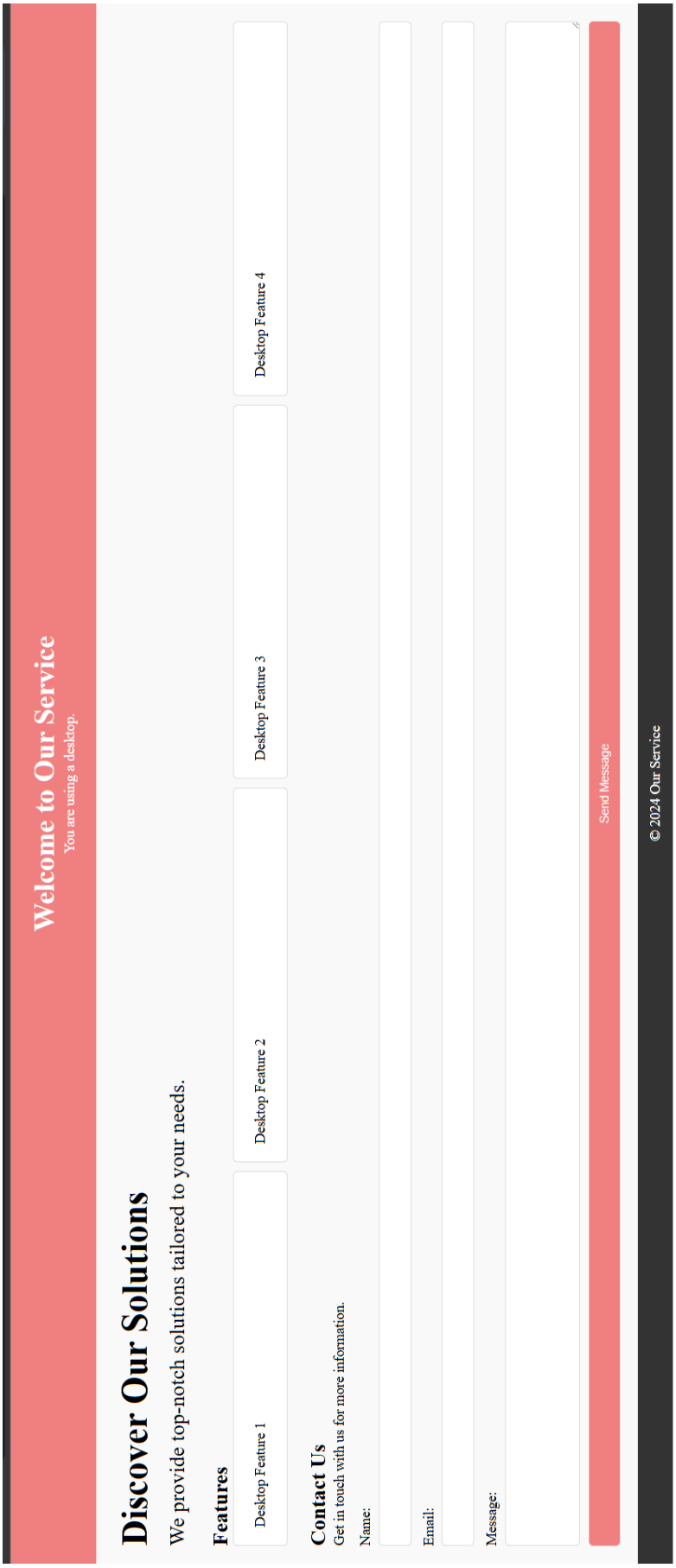

Figure 11: Adaptive Landing Page for Desktop Devices: Output

5 Cycle 2 Program 1
Dynamically Generated Content with JavaScript

Problem Statement

Create a web page that dynamically generates content using JavaScript. The page should include a button that, when clicked, generates a new random number and displays it on the page. Use JavaScript to manipulate the Document Object Model (DOM) to add and remove elements.

Expected Output

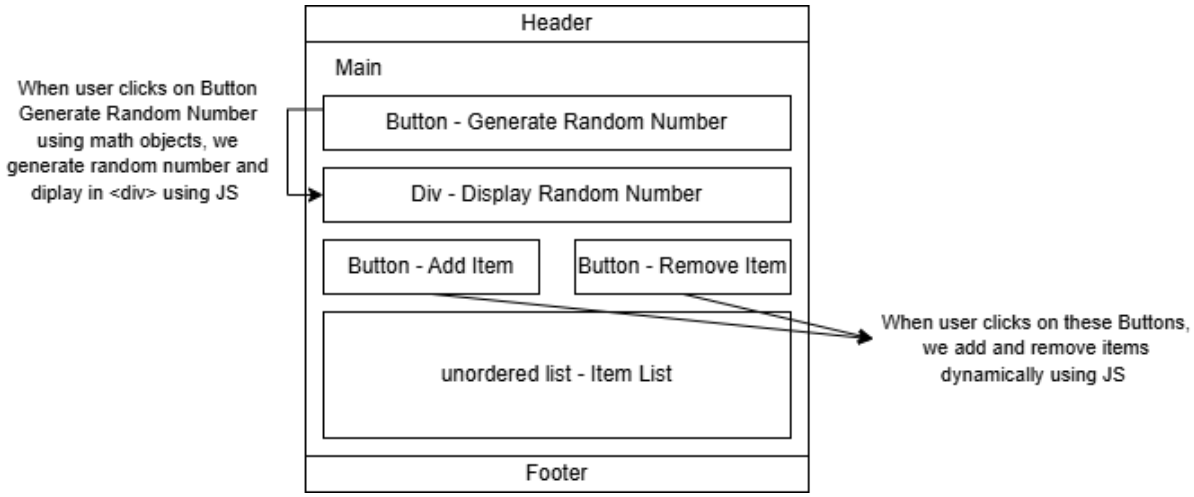

Figure 12: Dynamically Generated Content with JavaScript: Expected Output

Code

index.html

```
<html>
    <head>
        <title>Random Number Generator</title>
        <link rel="stylesheet" href="style.css">
        <script src="script.js"></script>
    </head>
    <body>
    <header><h1>Welcome to the Random Number Generator</h1></header>
        <main>
        <button onclick="generateRandomNumber()">Generate Random
           Number</button>
        <div id="randomNumber"></div>
        <button id="addItem" onClick="addItem()">Add Item</button>
        <button id="removeItem" onClick="removeItem()">Remove
           Item</button>
        <ul id="itemList"></ul>
        </main>
        <footer>&copy; 2025 Random Number Generator</footer>
```

```
    </body>
</html>
```

style.css

```
*{
    margin: 0;
    padding: 0;
    box-sizing: border-box;
}
header,footer{
    background-color: lightcoral;
    color: white;
    padding: 10px;
    text-align: center;
}
main{
    padding: 20px;
    text-align: center;
    background-color: lightblue;
}
button{
    padding: 10px;
    background-color: lightgreen;
    border: none;
    border-radius: 5px;
    cursor: pointer;
}
#randomNumber{
    font-weight: bold;
    margin: 20px;
}
```

script.js

```
function generateRandomNumber() {
    const randomNumber = Math.floor(Math.random() * 100) + 1;
    const numDisplay = document.getElementById("randomNumber");
    numDisplay.innerHTML= "";
    const newPara=document.createElement("p");
    newPara.innerHTML='Random Number: ' + randomNumber;
    numDisplay.appendChild(newPara);
}
function addItem() {
    var itemList = document.getElementById("itemList");
    var newItem = document.createElement("li");
    newItem.innerHTML = "Item " + (itemList.children.length + 1);
    itemList.appendChild(newItem);
}
function removeItem() {
    var itemList = document.getElementById("itemList");
    if (itemList.children.length > 0) {
        itemList.removeChild(itemList.lastChild);
    }
}
```

Output Screens

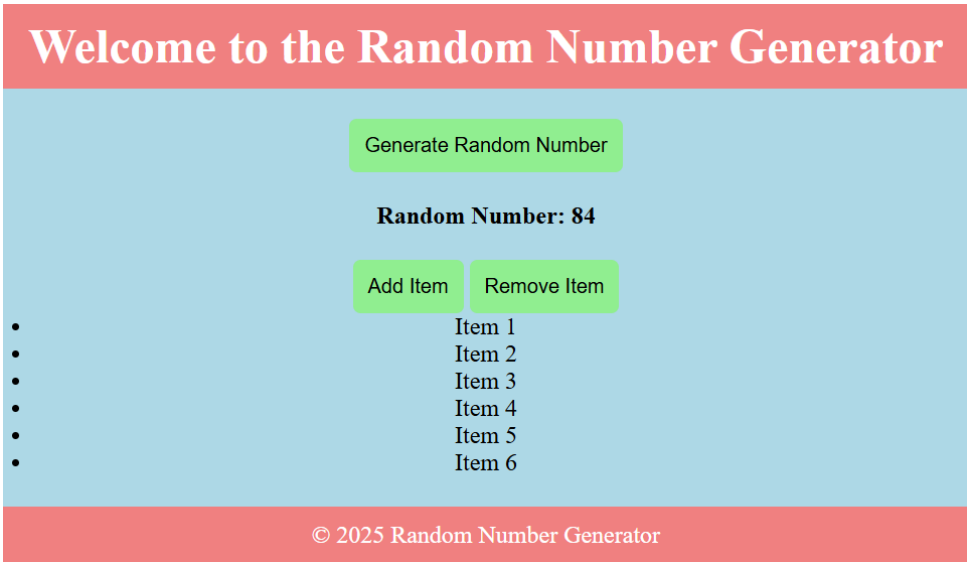

Figure 13: Dynamically Generated Content with JavaScript: Output

6 Cycle 2 Program 2
Interactive Shopping Cart with JavaScript

Problem Statement

Develop an interactive shopping cart using JavaScript. The cart should allow users to add and remove items, update quantities, and calculate the total price. Utilize JavaScript arrays and objects to store product information and manage cart operations.

Expected Output

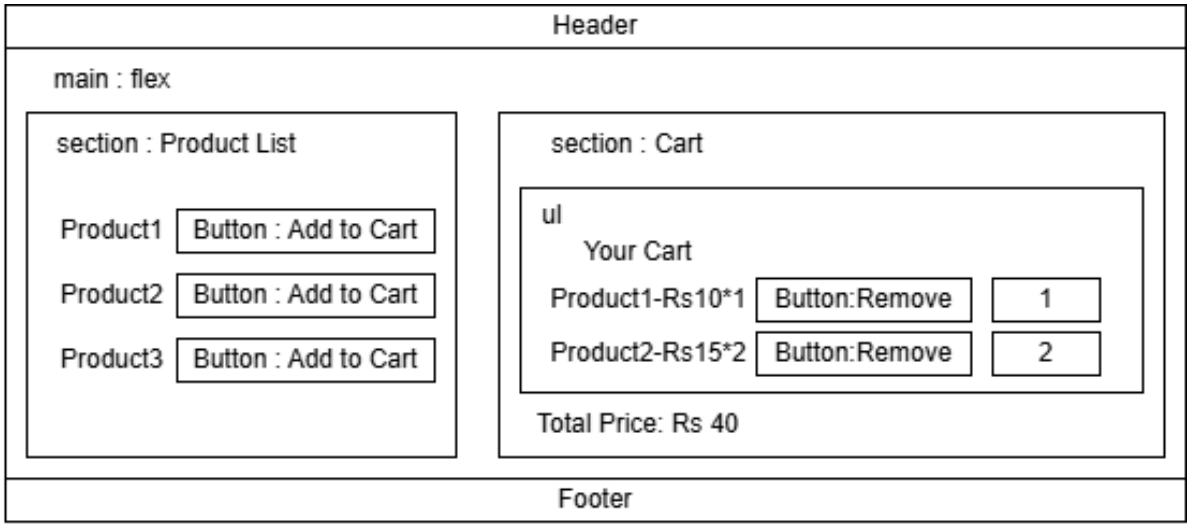

Figure 14: Interactive Shopping Cart with JavaScript: Expected Output

Code

index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width,
       initial-scale=1.0">
    <title>Shopping Cart</title>
    <link rel="stylesheet" href="styles.css">
</head>
<body>
    <header>
        <h1>Sample Shopping Cart</h1>
    </header>
    <main>
        <section id="product-list">
            <h2>Products</h2>
            <div class="product">
                <span>Product 1</span>
```

```
                <button onclick="addToCart('Product 1', 10)">Add to
                    Cart (Rs.10)</button>
            </div>
            <div class="product">
                <span>Product 2</span>
                <button onclick="addToCart('Product 2', 15)">Add to
                    Cart (Rs.15)</button>
            </div>
            <div class="product">
                <span>Product 3</span>
                <button onclick="addToCart('Product 3', 20)">Add to
                    Cart (Rs.20)</button>
            </div>
        </section>
        <section id="cart">
            <h2>Your Cart</h2>
            <ul id="cartItems">
                <!-- Cart items will be added here -->
            </ul>
            <p id="totalPrice">Total Price: Rs.0</p>
        </section>
    </main>
    <footer>
        <p>&copy; 2024 Shopping Cart Example</p>
    </footer>
    <script src="script.js"></script>
</body>
</html>
```

style.css

```
/* Basic reset */
* {
    margin: 0;
    padding: 0;
    box-sizing: border-box;
}

/* Styling for the header */
header {
    background-color: lightcoral;
    color: white;
    padding: 20px;
    text-align: center;
}

/* Styling for the main content */
main {
    padding: 20px;
    display: flex;
    justify-content: space-around;
    background-color: #f9f9f9;
}

/* Styling for product and cart sections */
#product-list, #cart {
    width: 45%;
```

```
}

.product, #cartItems li {
    background-color: white;
    padding: 10px;
    margin-bottom: 10px;
    border: 1px solid #ddd;
    border-radius: 5px;
}

button {
    padding: 10px;
    background-color: lightcoral;
    border: none;
    border-radius: 5px;
    color: white;
    cursor: pointer;
}

button:hover {
    background-color: darkred;
}

/* Styling for the total price */
#totalPrice {
    font-weight: bold;
}

/* Styling for the footer */
footer {
    text-align: center;
    padding: 10px;
    background-color: #333;
    color: white;
}
```

script.js

```
// Array to hold cart items
let cart = [];

// Function to add item to cart
function addToCart(name, price) {
    // Check if item already exists in cart
    let item = cart.find(item => item.name === name);
    if (item) {
        // If item exists, increase quantity
        item.quantity += 1;
    } else {
        // If item does not exist, add new item
        cart.push({ name, price, quantity: 1 });
    }
    updateCart();
}

// Function to remove item from cart
function removeFromCart(name) {
```

```
    // Filter out the item to be removed
    cart = cart.filter(item => item.name !== name);
    updateCart();
}

// Function to update item quantity
function updateQuantity(name, quantity) {
    let item = cart.find(item => item.name === name);
    if (item) {
      //  item.quantity = quantity;
        item.quantity =parseInt(quantity);
        if (item.quantity <= 0) {
            removeFromCart(name);
        } else {
            updateCart();
        }
    }
}

// Function to update the cart display
function updateCart() {
    let cartItems = document.getElementById('cartItems');
    cartItems.innerHTML = ''; // Clear existing items
    // Add each item to the cart display
    cart.forEach(item => {
        let li = document.createElement('li');
        li.innerHTML = `${item.name} - Rs.${item.price} x
           ${item.quantity}
                        <button
                           onclick="removeFromCart('${item.name}')">Remove</button
                        <input type="number" value="${item.quantity}"
                           min="0"
                        onchange="updateQuantity('${item.name}',
                           this.value)">`;
        cartItems.appendChild(li);
    });
    // Update total price
    let totalPrice = cart.reduce((total, item) => total + item.price *
       item.quantity, 0);
    document.getElementById('totalPrice').innerHTML = `Total Price:
       Rs.${totalPrice}`;
}
```

Output Screens

Figure 15: Interactive Shopping Cart with JavaScript: Output

7 Cycle 2 Program 3
Regular Expression-Based Text Manipulation

Problem Statement

Build a web application that performs text manipulation using regular expressions. The application should allow users to enter a text string and provide options for search, replace, and formatting. Implement regular expression patterns to identify and modify specific text elements.

Expected Output

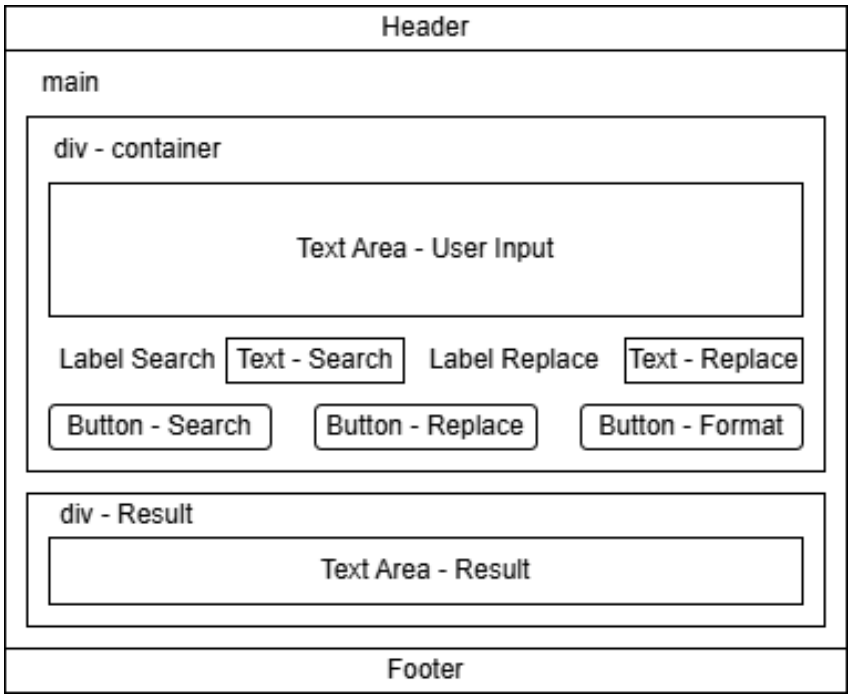

Figure 16: Regular Expression-Based Text Manipulation: Expected Output

Code

index.html

```
<html>
    <head>
        <title>Text Manipulation with RegEx</title>
        <link rel="stylesheet" href="style.css">
        <script src="script.js"></script>
    </head>
    <body>
        <header>
            <h1>Text Manipulation with RegEx</h1>
        </header>
        <main>
            <div class="container">
```

```
                <textarea id="inputText" placeholder="Enter your text
                    here..." rows="10" cols="50"></textarea>
                <br/>
                <label for="searchText">Search for:</label>
                <input type="text" id="searchText" placeholder="Enter
                    search term">
                <label for="replaceText">Replace with:</label>
                <input type="text" id="replaceText" placeholder="Enter
                    replacement text">
                <br/>
                <button onclick="performSearch()">Search</button>
                <button onclick="performReplace()">Replace</button>
                <button onclick="performFormat()">Format</button>
                <br/>
                <div id="results">
                    <h2>Results</h2>
                    <textarea id="outputText" placeholder="Result will
                        be displayed here..." rows="10" cols="50"
                        readonly></textarea>
                </div>
            </div>
        </main>
        <footer>
            <p>&copy; 2025 Text Manipulation with RegEx. All rights
                reserved.</p>
        </footer>
    </body>
</html>
```

style.css

```
/* Basic reset */
* {
    margin: 0;
    padding: 0;
    box-sizing: border-box;
}

/* Styling for the header */
header,footer {
    background-color: lightcoral;
    color: white;
    padding: 20px;
    text-align: center;
}

/* Styling for the main content */
main {
    padding: 20px;
    text-align: center;
    background-color: lightblue;
}

/* Styling for the button */
button {
    background-color: lightseagreen;
    border-radius: 10px;
```

```
    border: none;
    padding: 10px 20px;
    cursor: pointer;
    margin: 50px;
}
button:hover { background-color: greenyellow;}
/*Styling TextArea and Input Fields*/
textarea,input{
    padding: 10px;
    margin: 10px;
    border: 1px solid #ccc;
    border-radius: 5px;
    width:100%
}
```

script.js

```
function performSearch(){
    var inputText=inputText =
       document.getElementById("inputText").value;
    var searchText = document.getElementById("searchText").value;
    var resultText= document.getElementById("outputText");
    try{
        var matches=inputText.match(new RegExp(searchText,"g"));
        resultText.value=matches?matches.join("\n"):"No matches found";
    }
    catch(error){
        resultText.value="Invalid regular expression";
    }
}
function performReplace(){
    var inputText=inputText =
       document.getElementById("inputText").value;
    var searchText = document.getElementById("searchText").value;
    var replaceText = document.getElementById("replaceText").value;
    var resultText= document.getElementById("outputText");
    try{
        resultText.value=inputText.replace(new
           RegExp(searchText,"g"),replaceText);
    }
    catch(error){
        resultText.value="Invalid regular expression";
    }
}
function performFormat(){
    var inputText=inputText =
       document.getElementById("inputText").value;
    var resultText= document.getElementById("outputText");
    resultText.value=inputText.replace(/\b\w/g,char=>char.toUpperCase());
}
```

Output Screens

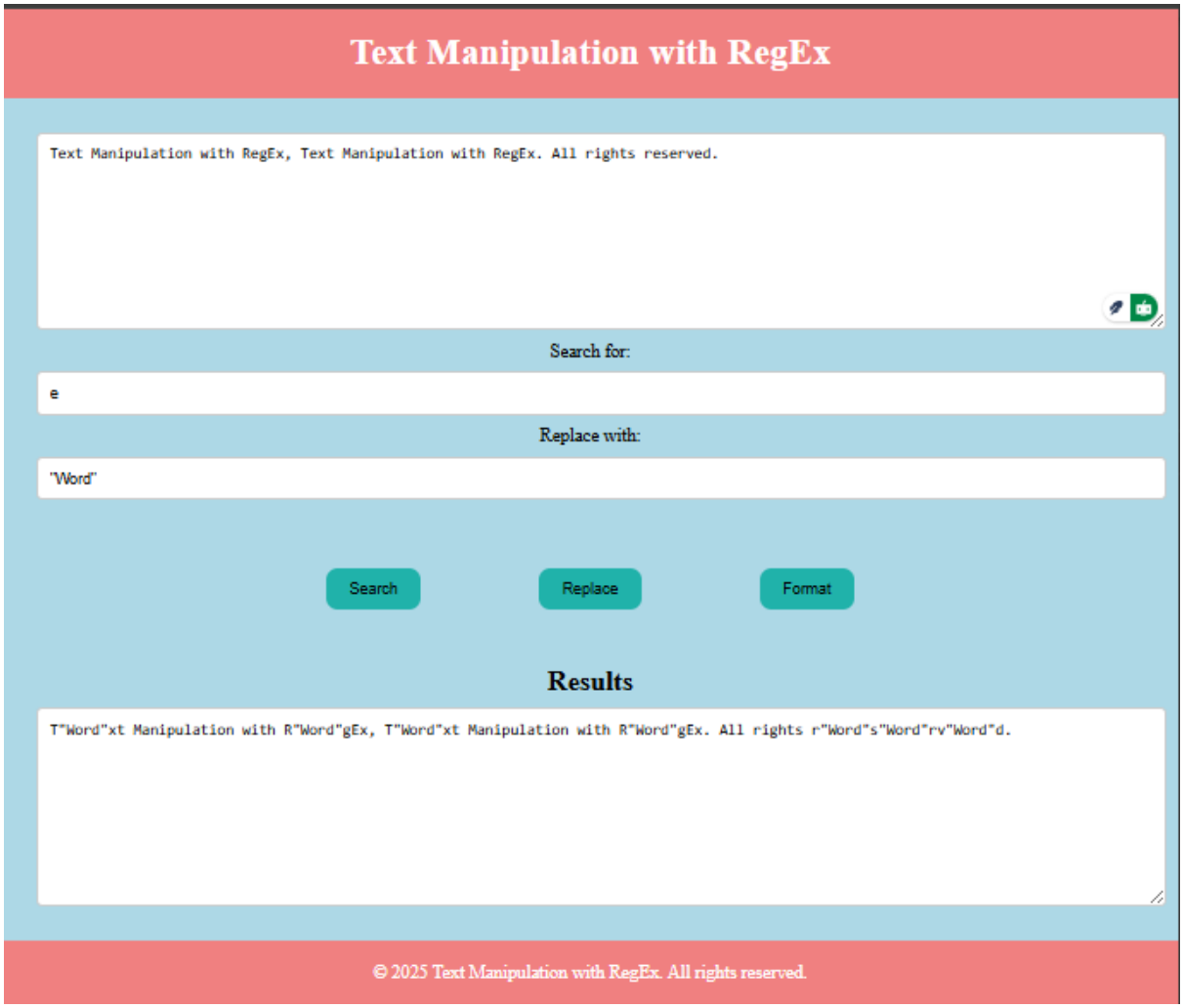

Figure 17: Regular Expression-Based Text Manipulation: Output

8 Cycle 2 Program 4
Async Data fetching and Display with JS promises and Async/await.

Problem Statement

Create a web page that fetches data from an API asynchronously using js promises and Async / wait.The page should display a loading indicator while the data is retrieved and then render the data in alist or table. Demonstrate the use of promises to handle asynchronous operations and improve the readability of the code. link:https://jsonplaceholder.typicode.com/ to fetch the data

Expected Output

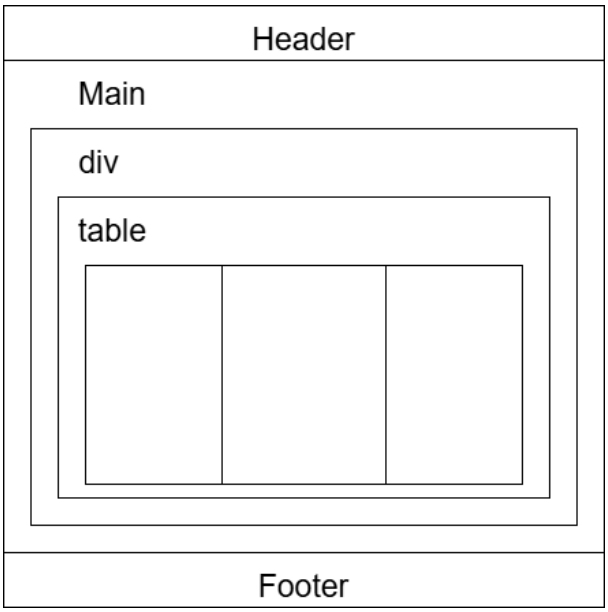

Figure 18: Asynchronous Data fetching and Display : Expected Output

Code

index.html

```
<html>
    <head>
        <meta charset="utf-8">
        <title>Asynchronous Data fetching</title>
        <script src="script.js"></script>  <!--External JS-->
        <link rel="stylesheet" href="style.css"> <!--External CSS-->
    </head>
    <body>
        <header>
            <h1>Asynchronous Data Fetching</h1>
        </header>
```

```
        <main>
            <div id="loading">Loading..</div>
            <table id="data">
                <thead>
                    <th>ID</th>
                    <th>Name</th>
                    <th>Email</th>
                </thead>
                <tbody id="dataBody">
                    <!-- data is dynamically added -->
                </tbody>
            </table>
        </main>
        <footer>
            &copy; 2025 Asynchronous Data Fetching
        </footer>
        <script src="script.js"></script>
    </body>
</html>
```

style.css

```
*{
    margin: 1;
    padding: 1;
    box-sizing: border-box;
}

header,footer{
    text-align: center;
    background-color: lightcoral;
    color: #fff;
    padding: 10px;
}
main{
    padding: 10px;
    display: flex;
    justify-content: space-around;
    background-color: lightblue;
}
#loading{
    font-size:20px;
    font-weight:bold;
    margin:20px;
}
table{
    border-collapse:collapse;
    width:100%;
    margin-top:20px;
}
th,td{
    padding: 8px;
    text-align: left;
    border: 1px solid #ddd;
}
th{
    background-color: #f2f2f2;
```

```
    color: black;
}
```

script.js

```
async function fetchData(){
    try{
        //show loading indicator
        document.getElementById('loading').style.display='block';
        //fetch dat a from api
        const response=await
           fetch('https://jsonplaceholder.typicode.com/users')
        .then(response => {
            if(!response.ok){
                return Promise.reject('Something went wrong');
            }
            return response.json();


        })
        .then(users=>{
            renderData(users);
        })
        .catch(error=>{
            console.error('There was a problem fetching the
               data',error);
            alert("failed to fetch data");
        })
        ;
    }
    catch (error){
        console.error('Error fetching data:',error);
    }
    finally{
        document.getElementById('loading').style.display='none';
    }
}
function renderData(data){
    const dataBody=document.getElementById('dataBody');
    dataBody.innerHTML='';
    data.forEach(user => {
        const row=document.createElement('tr');
        row.innerHTML =`
        <td>${user.id}</td>
        <td>${user.name}</td>
        <td>${user.email}</td>
        `;
        dataBody.appendChild(row);
    });
}
fetchData();
```

Output Screens

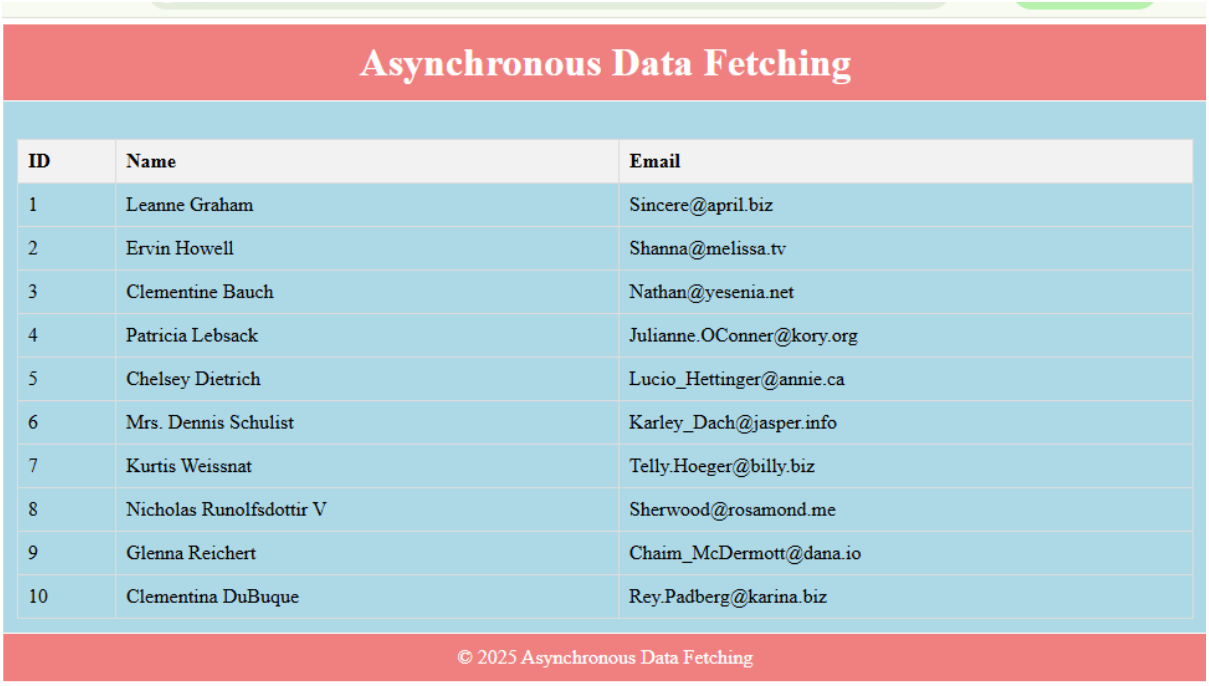

Figure 19: Asynchronous Data fetching and Display with JS promises and Async/await.: Output

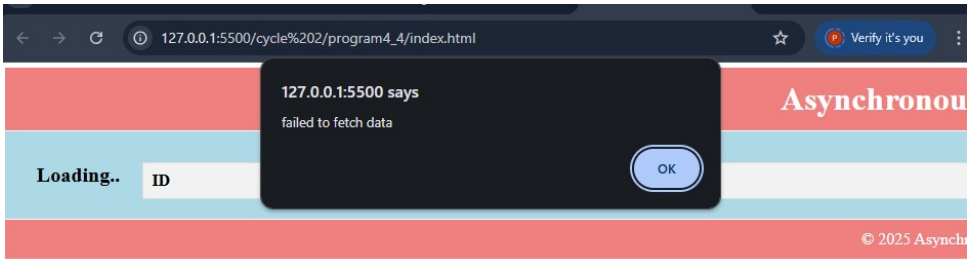

Figure 20: Asynchronous Data fetching and Display with JS promises and Async/await.: Output

9 Cycle 3 Program 1 Implementing Event-Driven Programming with Node.js

Problem Statement

Develop an event-driven Node.js application that utilizes the Node.js EventEmitter API. Create custom events to represent different occurrences, such as user actions or sensor readings. Register event listeners to handle these events and perform appropriate actions. Explore using modules like async/await to manage asynchronous event handling.

Technologies Used

- Node.js (v18 or later)

Setup Instructions

1. Create a new project directory: **mkdir sensor-events**
2. Navigate to the directory: **cd sensor-events**
3. Initialize a new Node.js project: **npm init -y**
    - This command initializes a new Node.js project with default settings.
    - npm init : This command is used to create a package.json file in the current directory, which is used to manage dependencies and store metadata about the project.
    - -y: This flag tells npm to use default settings for the project, without prompting the user for input.
    - The package.json file created by npm init -y provides a basic set of metadata for the project, including its name, version, description, and entry point. It also defines a set of scripts and specifies the license under which the project is released.
4. Create **index.js** file.
    - This file is a JavaScript file that serves as the entry point of a Node.js application. It's the file that's executed when you run the application using the node command.

File Structure

```
sensor-events/
    |-- index.js
    |-- package.json
    |-- package-lock.json
```

Database Design

- This project does not utilize a database.

Backend Development

index.js

- Custom Events
  - We'll create an **EventEmitter** instance to emit events like **userLogin**, and **sensorReading**.

```
const EventEmitter = require('events');
const customEmitter = new EventEmitter();
```

- Event Listeners
  - We'll register listeners to handle these events.

```
// Async listener for userLogin
customEmitter.on('userLogin', async (username) => {
    console.log(`User "${username}" is logging in...`);
    // Simulate a delay with async/await
    await simulateAsyncProcess('Checking user credentials...');
    console.log(`User "${username}" successfully logged in!`);
});

// Another async listener for sensor readings
customEmitter.on('sensorReading', async (sensorType, value) =>
   {
    console.log(`Received a reading from ${sensorType}:
       ${value}`);
    // Simulate an async processing step
    await simulateAsyncProcess(`Processing ${sensorType}
       data...`);
    if (sensorType === 'temperature' && value > 30) {
        console.log('Warning: Temperature is too high!');
    } else {
        console.log('Sensor data processed successfully.');
    }
});
```

- Asynchronous Event Handling
  - We'll use **async/await** to simulate asynchronous operations within event listeners.

```
async function simulateAsyncProcess(message) {
  return new Promise((resolve) => {
    setTimeout(() => {
      console.log(message);
      resolve();
    }, 2000); // Simulate a delay of 2 seconds
  });
}
```

- Simulated Sensor Data
  - We will create the functions for triggering events to simulate actions.

```
// Simulate a user logging
setTimeout(() => {
  customEmitter.emit('userLogin', 'john_doe');
}, 1000);

// Simulate a temperature sensor reading
setTimeout(() => {
  customEmitter.emit('sensorReading', 'temperature', 35);
}, 3000);

// Simulate a humidity sensor reading
setTimeout(() => {
  customEmitter.emit('sensorReading', 'humidity', 50);
}, 5000);
```

Frontend Development

- This is a backend-only application.

Testing and Debugging

- Testing Procedures
  - Manual testing by observing the console output.
  - Verify that events are emitted correctly and listeners are triggered.
  - Verify that **async/await** is working as expected.
- Debugging Techniques
  - **console.log()** statements to track event emissions and data.
  - Node.js debugger for step-by-step execution.
  - Verify that the asynchronous operation is executed.

Deployment

- This application can be deployed on any server that supports Node.js.

Output

```
C:\WT\Cycle3\Program1\sensor-events> node index.js
User "john_doe" is logging in...
Received a reading from temperature: 35
Checking user credentials...
User "john_doe" successfully logged in!
Received a reading from humidity: 50
Processing temperature data...
Warning: Temperature is too high!
Processing humidity data...
Sensor data processed successfully.
```

10 Cycle 3 Program 2 Working with Environment Variables and Dotenv in Node.js Apps

Problem Statement

Create a Node.js application that utilizes environment variables and dotenv to manage sensitive configuration data. Implement dotenv to load environment variables from a .env file and use them throughout the application. Demonstrate how to access and update environment variables securely.

Technologies Used

- Node.js (v18 or later)
- Express.js
- dotenv

Setup Instructions

1. Create a new project directory: **mkdir dotenv**
2. Navigate to the directory: **cd dotenv**
3. Initialize a new Node.js project: **npm init -y**
4. Install Express.js and dotenv: **npm install express dotenv**
5. Create **index.js** file.
6. Create **.env** file in the root directory.

File Structure

```
dotenv/
    |-- index.js
    |-- .env
    |-- package.json
    |-- package-lock.json
```

Database Design

- This project does not utilize a database.

Backend Development

Environment Variables:

- Create a `.env` file with key-value pairs for sensitive data.
- Example `.env` file:

```
PORT=3001
DB_HOST=localhost
DB_USER=myuser
API_KEY=your_secret_api_key
```

dotenv Configuration:

- Use `dotenv.config()` to load environment variables from the `.env` file.
- Example:

```
const dotenv = require('dotenv');
dotenv.config();
```

API Endpoints:

- `/`: A simple endpoint to display a welcome message.
- `/config`: An endpoint to display the loaded environment variables (for demonstration purposes only; **avoid this in production**).
- Example:
  - `/config` response:

```
{
    "dbHost": "localhost",
    "dbUser": "myuser",
    "apiKey": "your_secret_api_key"
}
```

Routing Logic:

- Route handlers for the `/` and `/config` endpoints.
- Error handling for undefined routes.

Error Handling:

- A 404 error handler for undefined routes.

index.js

```
const express = require('express');
const dotenv = require('dotenv');

dotenv.config();

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

app.get('/', (req, res) => {
    res.send('Environment Variables Demo!');
});

app.get('/config', (req, res) => {
    const dbHost = process.env.DB_HOST;
    const dbUser = process.env.DB_USER;
    const apiKey = process.env.API_KEY;

    res.json({
        dbHost,
        dbUser,
        apiKey
    });
});

app.use((req, res) => {
    res.status(404).send('Not Found');
});

app.listen(port, () => {
    console.log(`Server running on port ${port}`);
});
```

Frontend Development

- This is a backend-only application.

Testing and Debugging

Testing Procedures:

- **Using Postman:**
  1. Start the Node.js server.
  2. Send a GET request to `http://localhost:3001/config` (or the port specified in your `.env` file).
  3. Verify that the response contains the environment variables from the `.env` file.
  4. Send a GET request to a non-existent route to verify the 404 error handler.
- **Verify `.env` Loading:**
  - Change the values in the `.env` file, restart the server, and verify that the `/config` endpoint reflects the changes.

- **Verify Port Loading:**
  - Remove the `PORT` variable from the `.env` file. Verify that the server starts on port 3000.
  - Add the `PORT` variable back in and verify that the server starts on the port defined in the `.env` file.

Debugging Techniques:

- Use `console.log()` to check the values of `process.env` variables.
- Verify the `.env` file is in the correct location.
- Verify that `dotenv.config()` is called before accessing environment variables.

Deployment

- This application can be deployed on any server that supports Node.js.
- **.env** file should not be included in the production deployment.

Output

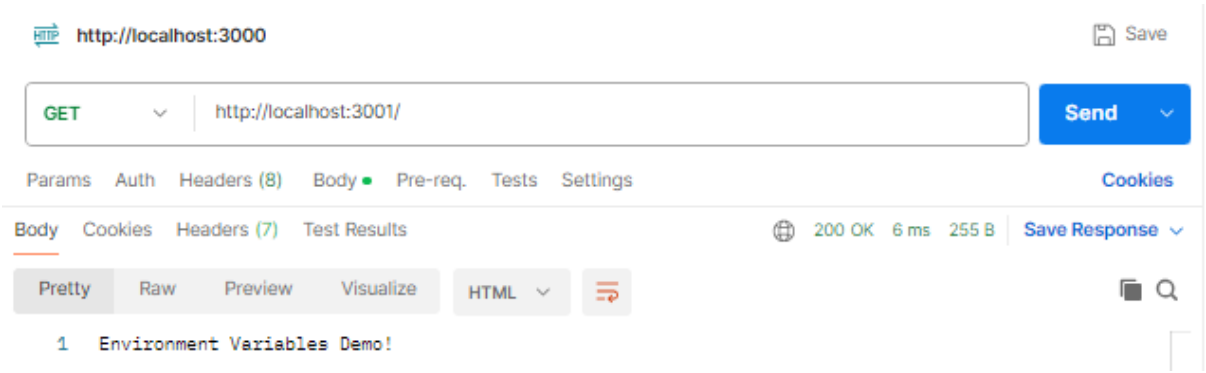

Figure 21: Output : Route handlers for the /

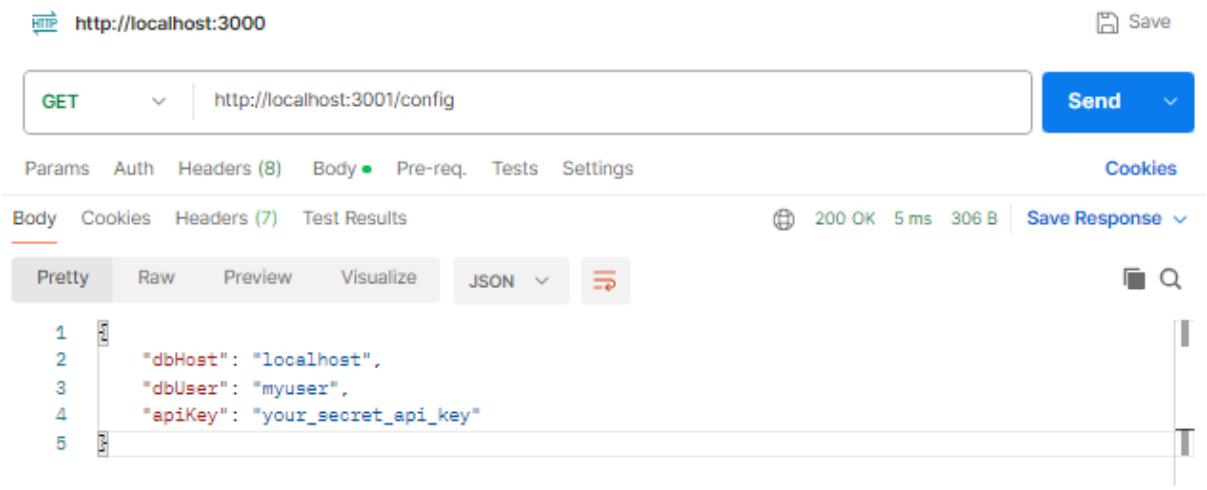

Figure 22: Output : Route handlers for the **/config**

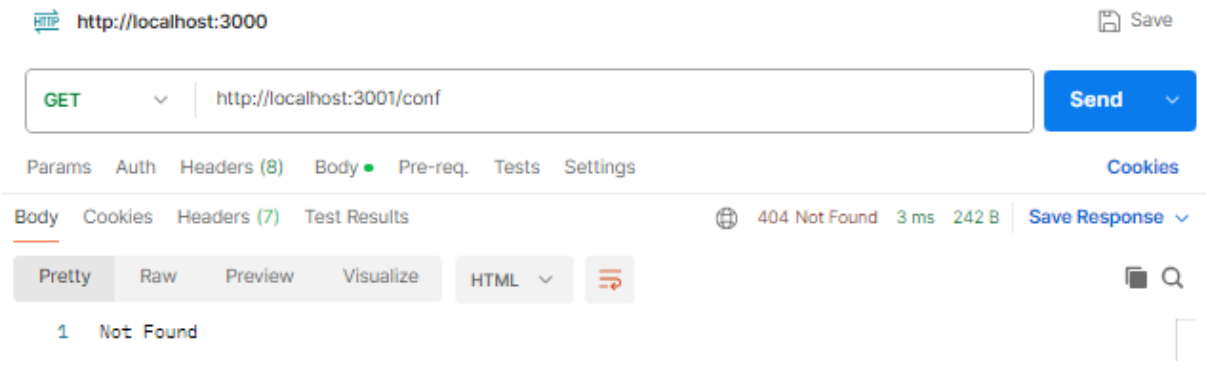

Figure 23: Output : non-existent route

11 Cycle 3 Program 3 Developing a Server-Side Application with Node.js and Express.js

Problem Statement

Build a server-side application using Node.js and Express.js. The application should provide an endpoint to accept user input, perform some processing, and return a response. Implement error handling and logging mechanisms to ensure the application runs reliably. Explore using Node.js modules for common tasks, such as file I/O and data validation.

Technologies Used

- Node.js (v18 or later)
- Express.js
- express-validator
- morgan

Setup Instructions

1. Create a new project directory: **mkdir express-user-data-processor**
2. Navigate to the directory: **cd express-user-data-processor**
3. Initialize a new Node.js project: **npm init -y**
4. Install Express.js, express-validator, and morgan:
   npm install express express-validator morgan
5. Create **index.js** file.

File Structure

```
express-user-data-processor/
    |-- index.js
    |-- package.json
    |-- package-lock.json
```

Database Design

- This project does not utilize a database.

Backend Development

API Endpoints

- **/process-input** (POST): Accepts user name and age, validates the input, saves it to a JSON file, and returns a response.
- Example:
  - Request body: **{ "name": "John Doe", "age": 30 }**
  - Response:
    { "message": "Data processed successfully",
    "data":{ "name": "John Doe", "age": 30, "timestamp": "ISO timestamp"}}
    or
    error message.

Middleware

- **express.json()**: For parsing JSON request bodies.
- **morgan('combined')**: For logging HTTP requests.
- **express-validator**: For input validation.

Routing Logic

- Route handler for the **/process-input** endpoint.

Processing Logic

- Extracts **name** and **age** from the request body.
- Creates a **userData** object with the extracted data and a timestamp.
- Saves the **userData** object to a **user-data.json** file.

Error Handling

- Uses **express-validator** to validate the request body.
- Handles file write errors and sends a 500 Internal Server Error response.
- Includes a general error handling middleware.

Logging

- Uses **morgan('combined')** to log HTTP requests.
- Logs file write errors to the console.

File I/O

- Uses the **fs** module to write user data to a JSON file.
- The file is located at **user-data.json** within the project directory.

index.js

```
const express = require('express');
const fs = require('fs');
const path = require('path');
const { body, validationResult } = require('express-validator');
const morgan = require('morgan');

const app = express();

app.use(express.json());
app.use(morgan('combined'));

app.post(
    '/process-input',
    [
        body('name').isString().withMessage('Name must be a string'),
        body('age').isInt({ min: 0 }).withMessage('Age must be a
           non-negative integer'),
    ],
    (req, res) => {
        const errors = validationResult(req);
        if (!errors.isEmpty()) {
            return res.status(400).json({ errors: errors.array() });
        }

        const { name, age } = req.body;
        const userData = { name, age, timestamp: new
           Date().toISOString() };
        const filePath = path.join(__dirname, 'user-data.json');

        fs.writeFile(filePath, JSON.stringify(userData, null, 2),
           (err) => {
            if (err) {
                console.error('Error writing to file:', err);
                return res.status(500).json({ message: 'Internal
                   Server Error' });
            }
            res.status(200).json({ message: 'Data processed
               successfully', data: userData });
        });
    }
);

app.use((err, req, res, next) => {
    console.error('Unhandled error:', err);
    res.status(500).json({ message: 'Something went wrong, please try
       again later' });
});

const PORT = process.env.PORT || 3000;
app.listen(PORT, () => {
    console.log(`Server is running on http://localhost:${PORT}`);
});
```

Frontend Development

- This is a backend-only application.

Testing and Debugging

- **Using Postman:**

1. **Open Postman:** Launch the Postman application.
2. **Create a New Request:** Click "New" and select "HTTP Request."
3. **Set the Method:** Change the request method to "POST."
4. **Enter the URL:** Enter the endpoint URL: `http://localhost:3000/process-input` (or the deployed URL).
5. **Set the Headers:**
   - Set the "Content-Type" header to "application/json."
6. **Set the Request Body:**
   - Go to the "Body" tab and select "raw" and then "JSON."
   - Enter the JSON request body:
     * **Valid Input:**

```
{
    "name": "Alice Smith",
    "age": 25
}
```

     * **Invalid Input Examples:**

```
{
    "name": 123, // Invalid name (not a string)
    "age": 25
}
```

```
{
    "name": "Alice Smith",
    "age": "invalid" // Invalid age (not an integer)
}
```

```
{
    "name": "Alice Smith",
    "age": -1 //invalid age (negative number)
}
```

```
{
    //missing name.
    "age": 25
}
```

```
{
    "name": "test",
    //missing age
}
```

7. **Send the Request:** Click the "Send" button.

8. **Verify the Response:**
   - Check the response status code (200 for success, 400 for bad request, 500 for server error).
   - Check the response body for the expected JSON output or error messages.
   - Verify the `user-data.json` file is created and contains the sent data.
9. **Test Error Scenarios:**
   - Send requests with invalid input to test validation errors.
   - Simulate file write errors (e.g., by changing file permissions) and test the 500 response.
10. **Verify Logging:** Check the console for logs generated by morgan.

- Testing with valid and invalid input for name and age.
- Testing file write error scenarios.
- Testing the general error handling middleware.

Debugging Techniques:

- `console.log()` statements for debugging.
- Node.js debugger for step-by-step execution.
- Checking the console for logged requests from morgan.
- Checking the console for file write errors.
- Testing the input validation.

Deployment Platform:

- This application can be deployed on any server that supports Node.js.

Recording Application Output:

`Postman Output:` Postman provides detailed output for each API request. The following has to be recorded:

- **HTTP Status Code:** The status code indicates the success or failure of the request.
- **Response Body (JSON):** The JSON response contains the data returned by the server.
- **Request Time:** The time taken to process the request.

Example Postman Output (Successful Request):

- **Status:** 200 OK
- **Body:**

```
{
    "message": "Data processed successfully",
    "data": {
        "name": "test",
        "age": 20,
        "timestamp": "2024-01-01T00:00:00.000Z"
    }
}
```

- **Time:** 100 ms

Example Postman Output (Validation Error):

- **Status:** 400 Bad Request
- **Body:**

```
{
    "errors": [
        {
            "location": "body",
            "msg": "Name must be a string",
            "path": "name",
            "type": "field"
        }
    ]
}
```

- **Time:** 50 ms

`File System Output:` Verify the contents of the `user-data.json` file. **Example File Contents (user-data.json):**

```
{
    "name": "test",
    "age": 20,
    "timestamp": "2024-01-01T00:00:00.000Z"
}
```

12 Cycle 4 Program 1 Building a Simple Web Server with HTTP Request Handling

Problem Statement

Create a basic HTTP server using Node.js and Express.js. The server should handle GET and POST requests, respond with appropriate HTTP status codes, and parse request payloads. Implement route handling to direct requests to specific functions for processing.

Technologies Used

- Node.js (v18 or later)
- Express.js

Setup Instructions

1. Create a new project directory: **mkdir Cycle4-Program1**
2. Navigate to the directory: **cd Cycle4-Program1**
3. Initialize a new Node.js project: **npm init -y**
4. Install Express.js :
   npm install express
5. Create **index.js and router.js** files.

File Structure

```
Cycle4-Program1/
    |-- index.js
    |-- router.js
    |-- package.json
    |-- package-lock.json
```

Backend Development

1. HTTP Methods
   - The example shows the use of different HTTP methods (GET and POST) and how they are handled. Understanding these methods is crucial for developing RESTful APIs, where different methods are used to perform different operations on resources.
2. Routing Logic
   - The code demonstrates how to define routes in an Express application. Routing is fundamental in backend development, as it determines how the server responds to different HTTP requests (GET, POST, etc.) at various endpoints.

3. Status Codes

    - The use of HTTP status codes (200 for success, 201 for resource creation) is important in communicating the outcome of a request to the client. Properly setting status codes is essential for building a reliable API.

index.js

```
const express = require('express'); // Import the Express framework
const app = express(); // Create an instance of the Express app

const router = require('./router'); // Import the router module

app.use('/api',router); // Use the router for all requests starting
    with '/api'

app.listen(3000, () => { // Start the server on port 3000
    console.log('Server started on port 3000');
});
```

router.js

```
const express = require('express');
const route = express.Router(); // Create a router instance

route.get('/', (req, res) => {
    res.status(200); // Set the response status to 200
    res.send('Hello, World! In GET'); // Send a response
});

route.post('/', (req, res) => {
    res.status(201); // Set the response status to 201
    res.send('Hello, World! In POST'); // Send a response
});

module.exports = route; // Export the router
```

Testing and Debugging

- **Using Postman:**

    1. **Open Postman:** Launch the Postman application.
    2. Verify response for a successful GET request.
    3. Verify response for a successful POST request.

Output Screens

Run the Command : **node index.js**
Output on command-line: Server started on port 3000

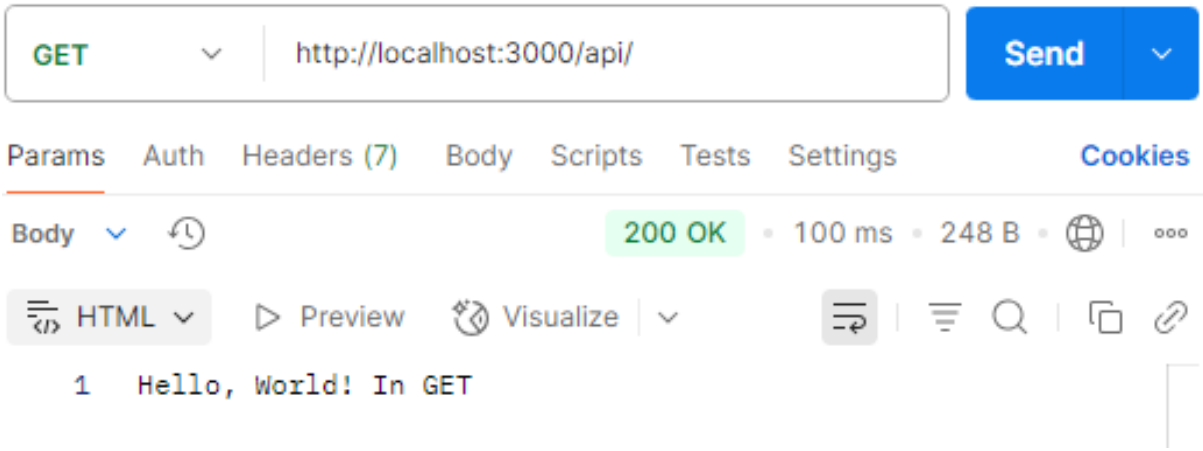

Figure 24: Verify response for a successful GET request: Output

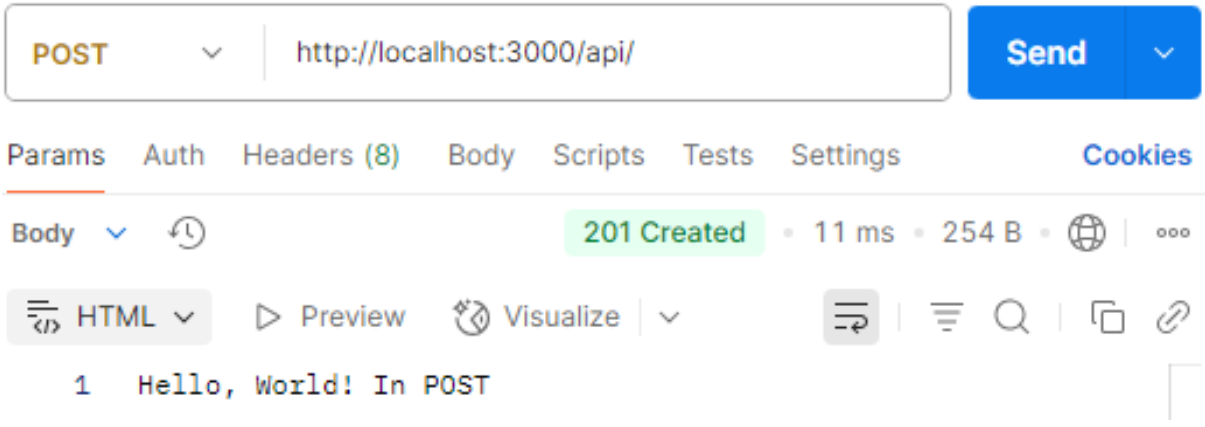

Figure 25: Verify response for a successful POST request.: Output

13 Cycle 4 Program 2 Designing and Implementing REST API for Resource Management.

Problem Statement

Design a REST API for managing a collection of resources such as books or products. Define the API end points for each resource operation (Create, Read, Update, Delete) and map them to HTTP Web (POST, GET, PUT, GET, DELETE). Implement the API using Node.js and Express.js including error handling and validation checks.

Technologies Used

- Node.js (v18 or later)
- Express.js
- body-parser
- express-validator

Setup Instructions

1. Create a new project directory: **mkdir Cycle4-Program2**
2. Navigate to the directory: **cd Cycle4-Program2**
3. Initialize a new Node.js project: **npm init -y**
4. Install required packages:
   npm install express body-parser express-validator
5. Create the following files and directories:
   index.js
   books.js
   route/
   route/bookRoute.js

File Structure

```
Cycle4-Program2/
    |-- index.js
    |-- books.js
    |-- route/
        |-- bookRoute.js
    |-- package.json
    |-- package-lock.json
```

Backend Development

1. HTTP Methods
    - The example shows the use of different HTTP methods (**GET**, **POST**, **PUT**, and **DELETE**) and how they are handled. Understanding these methods is crucial for developing RESTful APIs, where different methods are used to perform different operations on resources.
2. Routing Logic
    - The code demonstrates how to define routes in an Express application using ‘express.Router()‘. Routing is fundamental in backend development, as it determines how the server responds to different HTTP requests at various endpoints.
3. Data Validation
    - The ‘express-validator‘ library is used to validate incoming request data (e.g., ensuring IDs are positive integers, and book properties are not empty or have correct types). This is essential for maintaining data integrity and preventing common API errors.
4. Error Handling
    - The ‘handleValidationErrors‘ middleware centrally manages validation results, sending appropriate 400 (Bad Request) status codes if errors are found. Specific route handlers also include error handling for cases like ”Book not found” (404) and internal server errors (500).
5. Status Codes
    - The use of HTTP status codes (200 for success, 201 for resource creation, 204 for successful deletion with no content, 400 for bad request, 404 for not found, 500 for internal server error) is important in communicating the outcome of a request to the client. Properly setting status codes is essential for building a reliable API.

Code

index.js

```
const express = require('express');
//import module to parse JSON data
const bodyParser = require('body-parser');

const app = express();
app.use(bodyParser.json()); // Move this above the router middleware

const router = require('./route/bookRoute');
app.use('/books', router);

app.listen(3000, () => {
    console.log('Server is running on port 3000');
});
```

books.js

```
let books = [
    {id:1, name:"The Great Gatsby", author:"F. Scott
       Fitzgerald",price:19.99, quantity: 5},
    {id:2, name:"To Kill a Mockingbird", author:"Harper Lee", price:
       14.99, quantity: 3},

];
let nextId = 3;

module.exports = {books, nextId};
// The above code defines an array of book objects, each with
   properties such as id, name, author, price, and quantity.
```

route/bookRoute.js

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

const router = express.Router(); //create a new router instance

// install express validator to validate data
const { body, param, validationResult } = require('express-validator');

let { books, nextId } = require('../books'); //import books data from
   books.js

//validation rules
const validateIDParam = [
    param('id').isInt({ gt: 0 }).withMessage('Invalid ID')
];
const validateBook = [
    body('name').trim().notEmpty().withMessage('Name is required'),
    body('author').trim().notEmpty().withMessage('Author is required'),
    body('price').isFloat({ gt: 0 }).withMessage('Price must be a
       positive number'),
    body('quantity').isInt({ gt: 0 }).withMessage('Quantity must be a
       positive integer')
];

// ---Middleware to handle validation results---

const handleValidationErrors = (req, res, next) => {
    const errors = validationResult(req);
    if (!errors.isEmpty()) {
        return res.status(400).json({ errors: errors.array() });
    }
    next(); //proceed to the next middleware or route handler
};

router.get('/', (req, res) => {
    try {
        res.json(books); //send the books array as a JSON response
    } catch (error) {
        console.log(error);
        res.status(500).send('Internal Server Error'); //send a 500
           error response
```

```
    }
});
router.get('/:id', validateIDParam, handleValidationErrors, (req, res)
   => {
    const id = parseInt(req.params.id); //parse the id from the
       request parameters
    const book = books.find((b) => b.id === id); //find the book with
       the given id
    if (!book) {
        return res.status(404).send('Book not found'); //send a 404
           error response if not found
    }
    res.json(book); //send the book as a JSON response
});
router.post('/', validateBook, handleValidationErrors, (req, res) => {
    const newBook = { id: nextId++, ...req.body }; //create a new book
       object with the next id and request body
    books.push(newBook); //add the new book to the books array
    res.status(201).json(newBook); //send the new book as a JSON
       response with a 201 status code
});
router.put('/:id',validateIDParam,validateBook,handleValidationErrors,(req,res)=>{
    const id = parseInt(req.params.id);
    const index = books.findIndex(book=>book.id===id);
    if(index==-1)
    {
        return res.status(404).json({error: 'Book not found'});
    }
    const updatedBook={
        ...books[index],
        name:req.body.name,
        author:req.body.author,
        price:req.body.price,
        quantity:req.body.quantity

    };
    books[index] = updatedBook;
    res.json(updatedBook);

});

router.delete('/:id',validateIDParam,handleValidationErrors,(req,res)=>{
    const id = parseInt(req.params.id);
    const index = books.findIndex(book=>book.id===id);
    if(index==-1)
    {
        return res.status(404).json({error:"Book not found"});
    }
    books.splice(index,1);
    res.sendStatus(204);

});

module.exports = router; // Export the router
```

Output Screens

Run the Command : **node index.js**
Output on command-line: Server is running on port 3000

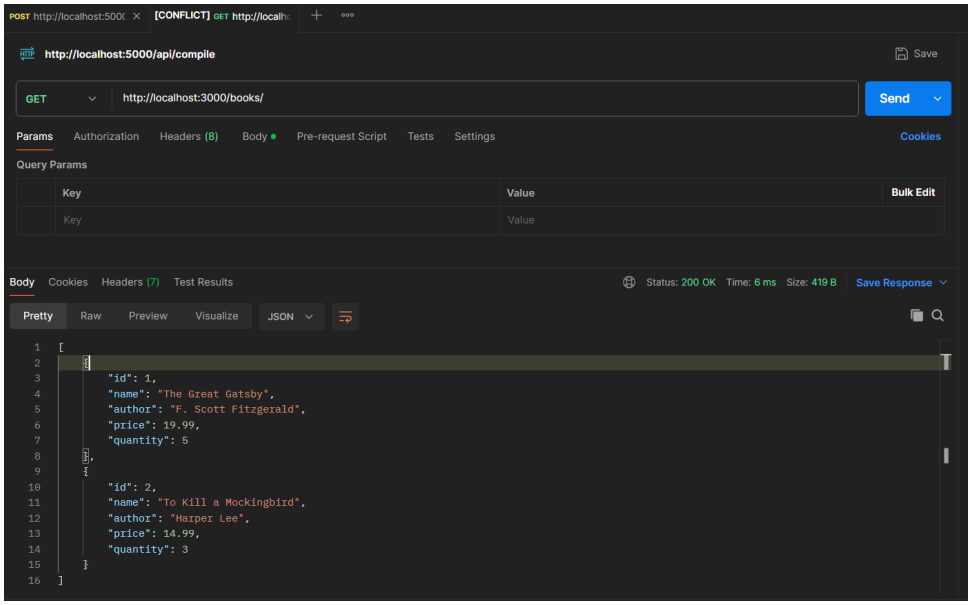

Figure 26: GET Method: Output

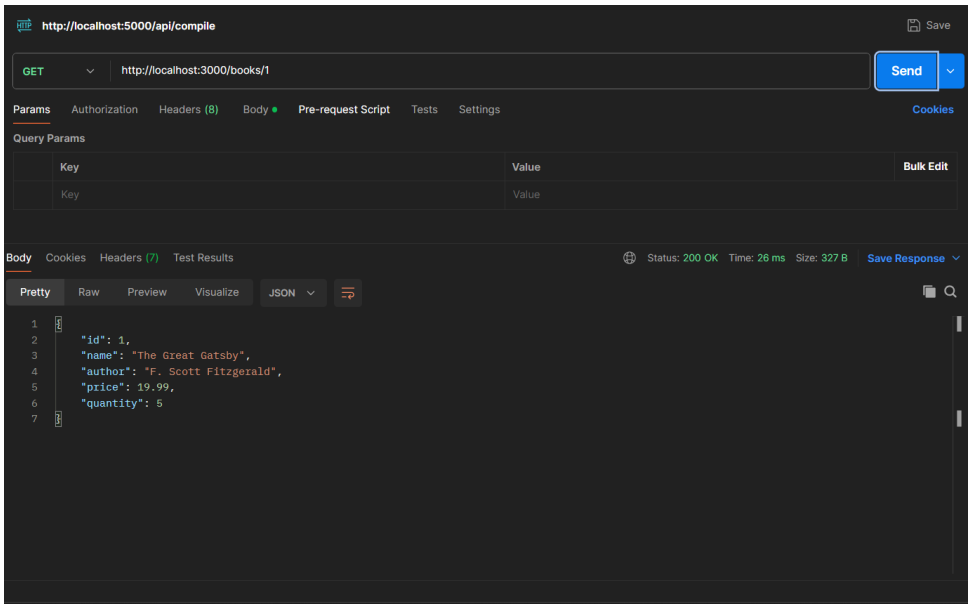

Figure 27: GET Method by ID: Output

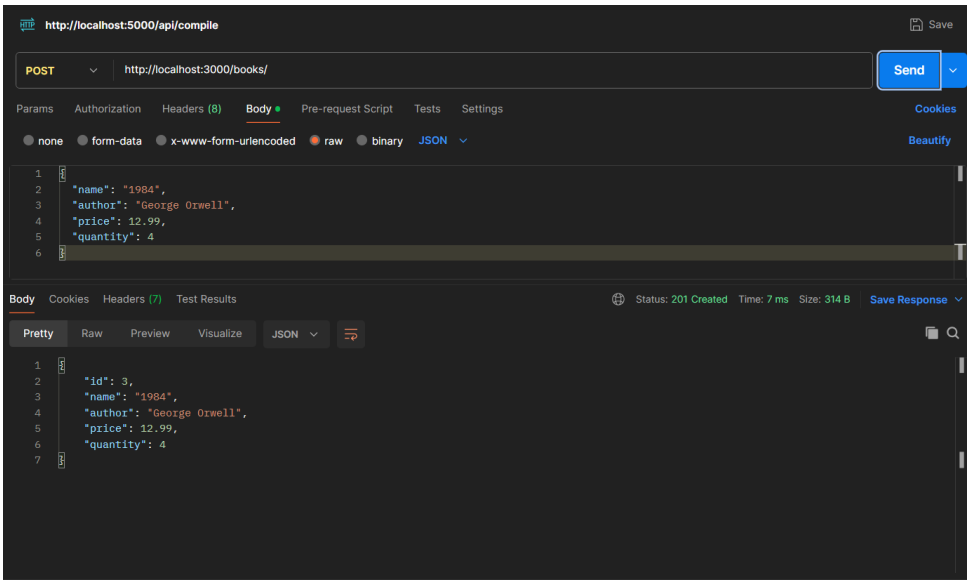

Figure 28: POST Method: Output 1

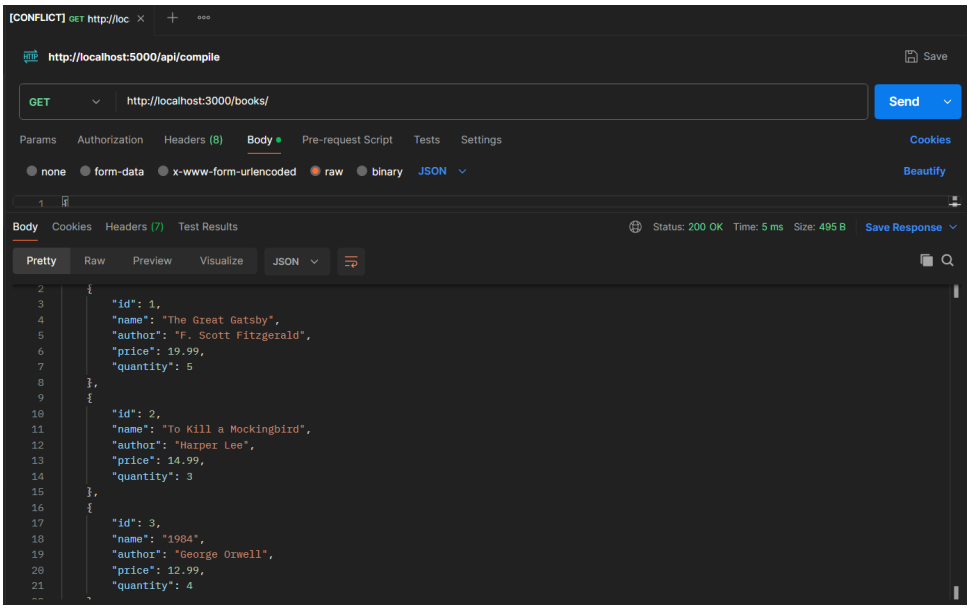

Figure 29: POST Method: Output 2

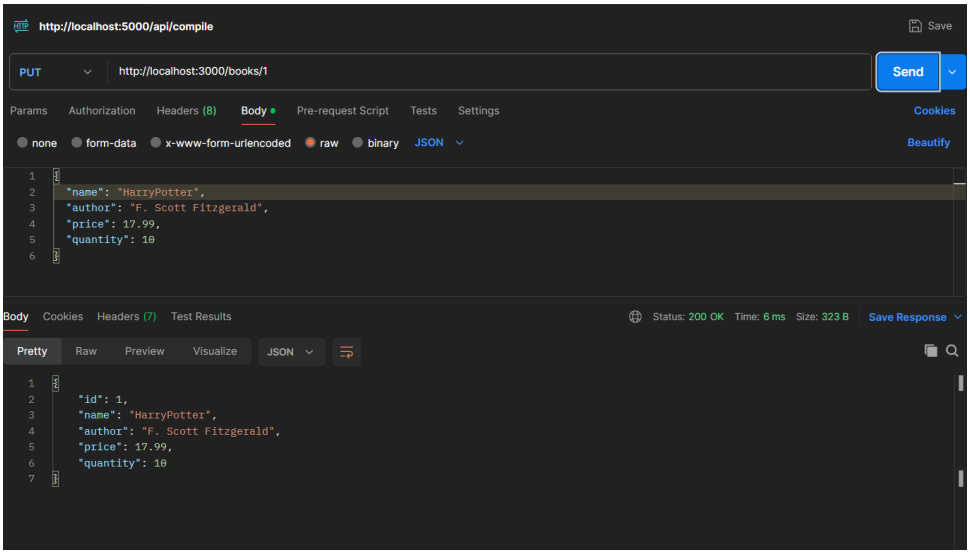

Figure 30: PUT Method: Output 1

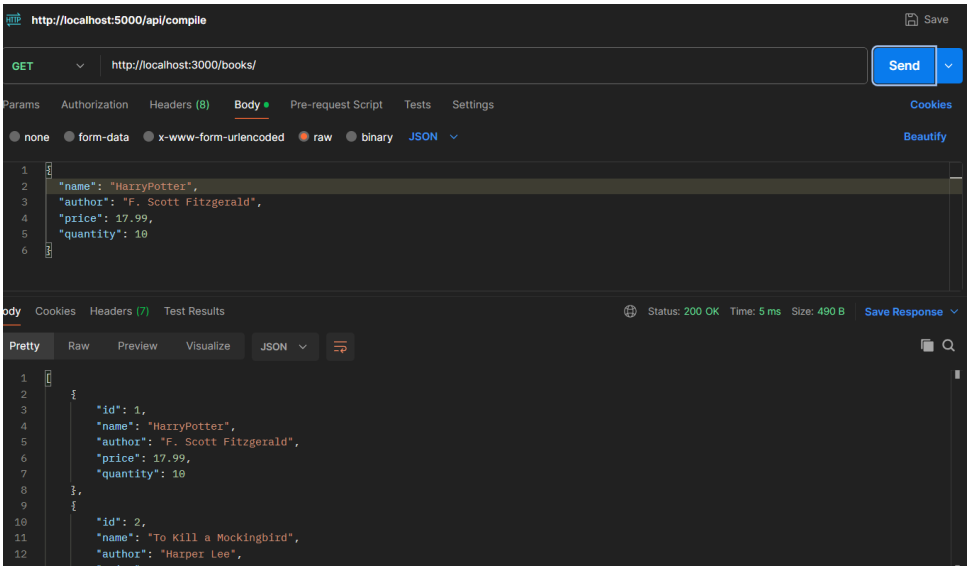

Figure 31: PUT Method: Output 2

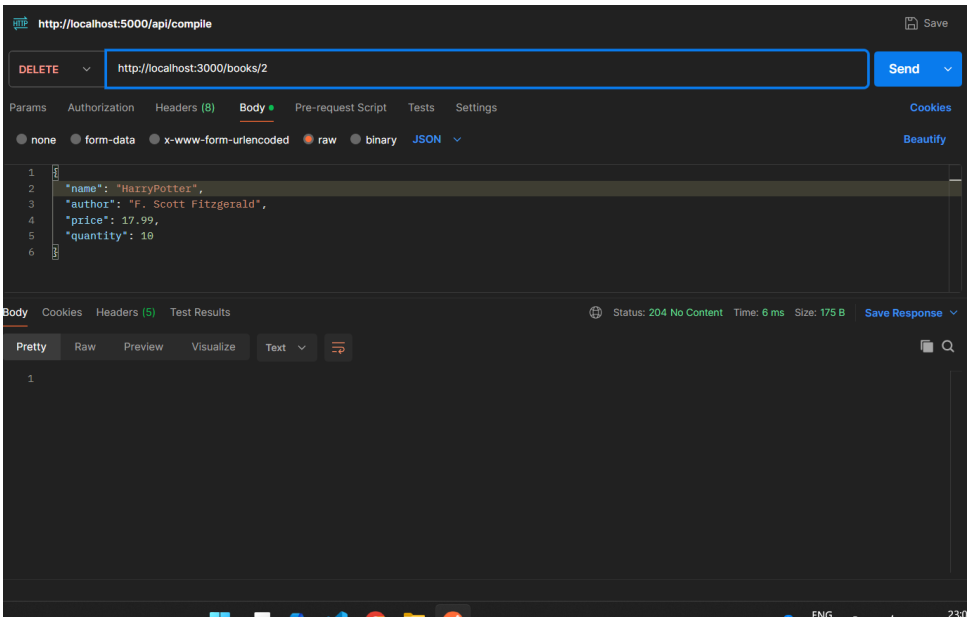

Figure 32: DELETE Method: Output 1

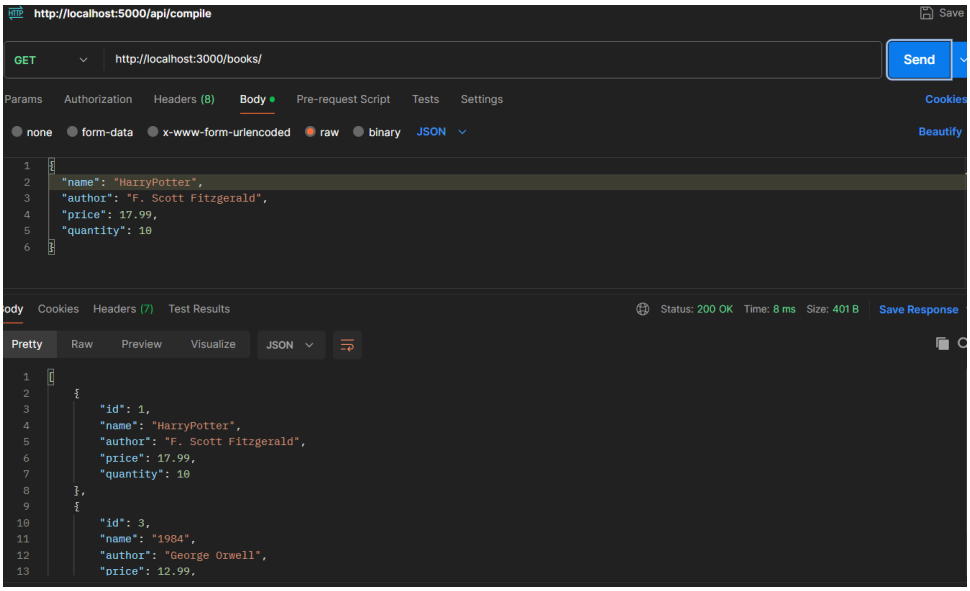

Figure 33: DELETE Method: Output 2

14 Cycle 5 Program 1 Developing a Full-Stack Web Application with Express and MongoDB

Problem Statement

Create a full-stack web application using Express.js and MongoDB. The application should allow users to create, read, update, and delete (CRUD) data stored in a MongoDB database. Implement the Express framework to handle routing, request processing, and templating. Utilize MongoDB to store and retrieve data using the Node.js MongoDB driver.

Technologies Used

- **Node.js** (v18 or later): A JavaScript runtime environment.
- **Express.js**: A fast, unopinionated, minimalist web framework for Node.js.
- **MongoDB**: A NoSQL database used for storing application data.
- **MongoDB Node.js Driver**: The official Node.js driver for MongoDB.
- **EJS (Embedded JavaScript templates)**: A simple templating language for generating HTML with plain JavaScript.
- **body-parser**: Node.js middleware for parsing incoming request bodies.

Setup Instructions

1. **Create a new project directory: mkdir crud-app-express-mongo**
2. **Navigate to the directory: cd crud-app-express-mongo**
3. **Initialize a new Node.js project: npm init -y**
4. **Install required packages:**
   npm install express mongodb body-parser ejs
5. **Create files and directories:**
   - Create the main application file: **main.js**
   - Create a directory for EJS views: **mkdir views**
   - Inside 'views', create the following template files: **views/index.ejs views/create.ejs views/edit.ejs**
6. **Ensure MongoDB is running:** Make sure your local MongoDB instance is running on 'mongodb://localhost:27017'.

File Structure

```
crud-app-express-mongo/
    |-- main.js
    |-- views/
    |   |-- index.ejs
    |   |-- create.ejs
    |   |-- edit.ejs
    |-- package.json
    |-- package-lock.json
```

Backend Development

1. **HTTP Methods**
    - This application implements full CRUD operations using appropriate HTTP methods:
        - **GET /**: Retrieves and displays all items from the database.
        - **GET /create**: Displays a form to create a new item.
        - **POST /create**: Handles the creation of a new item in the database.
        - **GET /edit/:id**: Displays a form pre-filled with data to edit an existing item.
        - **POST /edit/:id**: Handles the update of an existing item in the database.
        - **POST /delete/:id**: Handles the deletion of an item from the database.
    - Understanding these methods is crucial for developing RESTful APIs, where different methods are used to perform different operations on resources.
2. **Routing Logic**
    - The code demonstrates how to define routes in an Express application. Routing is fundamental in backend development, as it determines how the server responds to different HTTP requests at various endpoints.
3. **Database Integration**
    - The **MongoDB Node.js Driver** is used to establish a connection to a MongoDB database.
    - The application interacts with a collection named ‘items‘ to perform ‘insertOne‘, ‘find‘, ‘findOne‘, ‘updateOne‘, and ‘deleteOne‘ operations.
    - ‘ObjectId‘ from ‘mongodb‘ is used to correctly query and manipulate documents by their unique MongoDB ID.
4. **Templating with EJS**
    - **EJS** is configured as the view engine, allowing dynamic HTML generation.
    - Templates (‘index.ejs‘, ‘create.ejs‘, ‘edit.ejs‘) are rendered by Express routes, passing data from the backend to the frontend for display and form prepopulation.

5. **Error Handling**

   - Each route includes ‘try-catch‘ blocks to gracefully handle potential errors during database operations or rendering. Appropriate HTTP status codes (e.g., 500 for server errors) are sent back to the client.

   - The use of HTTP status codes (200 for success, 302 for redirects, 500 for internal server error) is important in communicating the outcome of a request to the client. Properly setting status codes is essential for building a reliable web application.

Code

main.js

```
// Import required modules
const express = require('express');
const { MongoClient, ObjectId } = require('mongodb');
// MongoClient is a class that allows us to connect to a MongoDB database
// ObjectId is a class that allows us to create a unique identifier for each item
const bodyParser = require('body-parser');
const path = require('path');
// we need to install ejs module, to load ejs views

// Create an Express app
const app = express();
const PORT = 3000;

// Middleware
app.use(bodyParser.urlencoded({ extended: true })); //used to parse URL-encoded data
app.use(bodyParser.json());
app.set('view engine', 'ejs');
app.set('views', path.join(__dirname, 'views'));

// MongoDB configuration
const MONGO_URI = 'mongodb://localhost:27017';
const DATABASE_NAME = 'SNIST';
let db;

// Connect to MongoDB
MongoClient.connect(MONGO_URI, { useNewUrlParser: true, useUnifiedTopology: true })
  .then(client => {
    console.log('Connected to MongoDB');
    db = client.db(DATABASE_NAME);
 })
  .catch(err => console.error(err));

// Routes

// Home route to list all items
app.get('/', async (req, res) => {
  try {
```

```
38      const items = await db.collection('items').find().toArray();
39      console.log(items);
40      res.render('index', { items });
41    } catch (error) {
42      res.status(500).send('Error fetching items');
43    }
44  });
45
46  // Form to create a new item
47  app.get('/create', (req, res) => {
48    res.render('create');
49  });
50
51  // Create a new item
52  app.post('/create', async (req, res) => {
53    try {
54      await db.collection('items').insertOne({ name: req.body.name,
          description: req.body.description });
55      res.redirect('/');
56    } catch (error) {
57      res.status(500).send('Error creating item');
58    }
59  });
60
61  // Form to edit an item
62  app.get('/edit/:id', async (req, res) => {
63    try {
64      const item = await db.collection('items').findOne({ _id: new
          ObjectId(req.params.id) });
65      res.render('edit', { item });
66    } catch (error) {
67      res.status(500).send('Error fetching item');
68    }
69  });
70
71  // Update an item
72  app.post('/edit/:id', async (req, res) => {
73    try {
74      await db.collection('items').updateOne(
75        { _id: new ObjectId(req.params.id) },
76        { $set: { name: req.body.name, description: req.body.description
            } }
77      );
78      res.redirect('/');
79    } catch (error) {
80      res.status(500).send('Error updating item');
81    }
82  });
83
84  // Delete an item
85  app.post('/delete/:id', async (req, res) => {
86   try {
87      await db.collection('items').deleteOne({ _id: new
          ObjectId(req.params.id) });
88      res.redirect('/');
89   } catch (error) {
90   res.status(500).send('Error deleting item');
91   }
```

```
});

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

views/index.ejs

(Note: You'll need to create this file in the 'views' directory)

```
<html<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width,
     initial-scale=1.0">
  <title>Items List</title>
  <link rel="stylesheet"
  href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0-alpha1/dist/css/bootstrap.min.
</head>
<body>
  <div class="container mt-5">
    <h1 class="mb-4">Items List</h1>
    <a href="/create" class="btn btn-primary mb-4">Add New Item</a>

    <% if (items.length > 0) { %>
      <table class="table table-bordered">
        <thead>
          <tr>
            <th>Name</th>
            <th>Description</th>
            <th>Actions</th>
          </tr>
        </thead>
        <tbody>
          <% items.forEach(item => { %>
            <tr>
              <td><%= item.name %></td>
              <td><%= item.description %></td>
              <td>
                <a href="/edit/<%= item._id %>" class="btn btn-warning
                   btn-sm">Edit</a>
                <form action="/delete/<%= item._id %>" method="POST"
                   style="display: inline;">
                  <button type="submit" class="btn btn-danger
                     btn-sm">Delete</button>
                </form>
              </td>
            </tr>
          <% }); %>
        </tbody>
      </table>
    <% } else { %>
      <p>No items found. Click "Add New Item" to create one.</p>
    <% } %>
  </div>
</body>
```

```
</html>
```

views/create.ejs

(Note: You'll need to create this file in the 'views' directory)

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width,
     initial-scale=1.0">
  <title>Create Item</title>
  <link rel="stylesheet"
     href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0-alpha1/dist/css/bootstrap.m
</head>
<body>
  <div class="container mt-5">
    <h1 class="mb-4">Create New Item</h1>
    <form action="/create" method="POST">
      <div class="mb-3">
        <label for="name" class="form-label">Name</label>
        <input type="text" class="form-control" id="name" name="name"
           required>
      </div>
      <div class="mb-3">
        <label for="description" class="form-label">Description</label>
        <textarea class="form-control" id="description"
           name="description" rows="4" required></textarea>
      </div>
      <button type="submit" class="btn btn-success">Create</button>
      <a href="/" class="btn btn-secondary">Cancel</a>
    </form>
  </div>
</body>
</html>
```

views/edit.ejs

(Note: You'll need to create this file in the 'views' directory)

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width,
     initial-scale=1.0">
  <title>Edit Item</title>
  <link rel="stylesheet"
     href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0-alpha1/dist/css/bootstrap.m
</head>
<body>
  <div class="container mt-5">
    <h1 class="mb-4">Edit Item</h1>
    <form action="/edit/<%= item._id %>" method="POST">
      <div class="mb-3">
        <label for="name" class="form-label">Name</label>
```

```
        <input type="text" class="form-control" id="name" name="name"
            value="<%= item.name %>" required>
      </div>
      <div class="mb-3">
        <label for="description" class="form-label">Description</label>
        <textarea class="form-control" id="description"
            name="description" rows="4" required><%= item.description
            %></textarea>
      </div>
      <button type="submit" class="btn btn-success">Save
          Changes</button>
      <a href="/" class="btn btn-secondary">Cancel</a>
    </form>
  </div>
</body>
</html>
```

Testing and Debugging

- **Accessing the Application:**
  1. **Start the server:** In your terminal, navigate to the ‘crud-app-express-mongo‘ directory and run: **node main.js**
  2. **Open your web browser:** Go to **http://localhost:3000**
  3. **Create items:** Click on ”Add New Item” to create new entries. Fill the form and submit.
  4. **Read items:** Observe the list of items displayed on the home page.
  5. **Update items:** Click ”Edit” next to an item, modify the fields, and submit.
  6. **Delete items:** Click ”Delete” next to an item to remove it.
- **Debugging:**
  - Check the terminal where your server is running for ‘console.log‘ messages or error output.
  - Use your browser’s developer tools (F12) to inspect network requests and console errors on the client side.
  - Ensure your MongoDB server is running and accessible on ‘localhost:27017‘. You can use a tool like MongoDB Compass to verify data in the ‘SNIST‘ database and ‘items‘ collection.

Output Screens

Run the Command : **node main.js**
Output on command-line:

```
Connected to MongoDB
Server running on http://localhost:3000
```

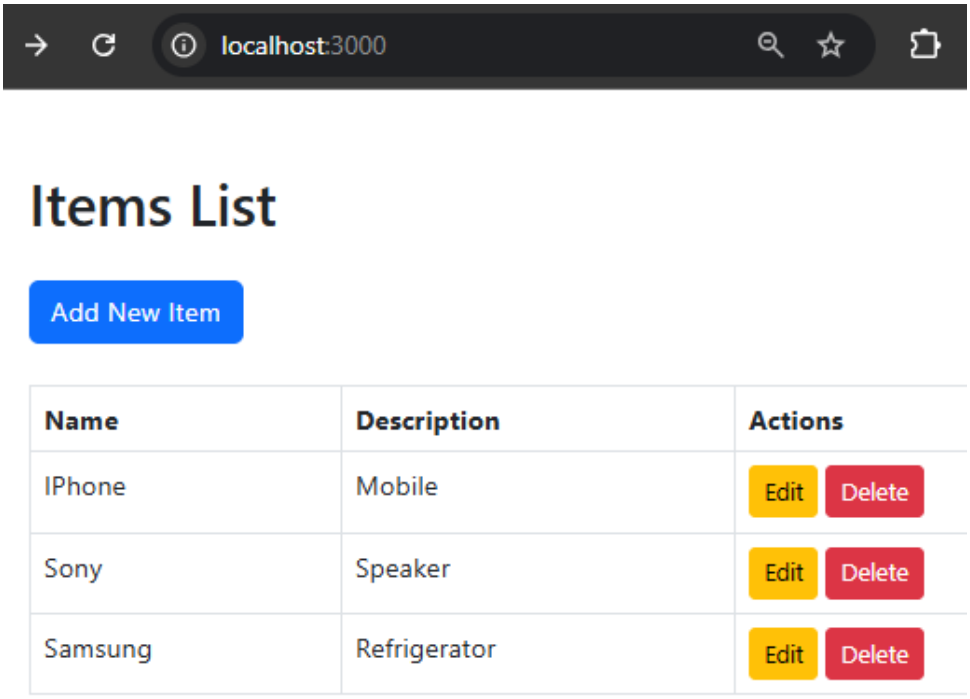

Figure 34: Home Page with Items

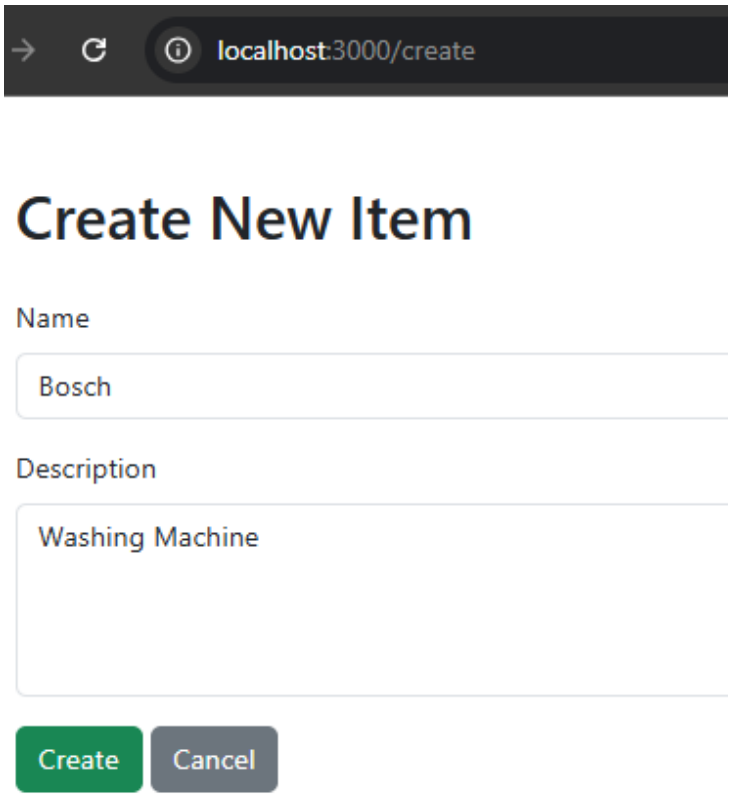

Figure 35: Create New Item Form

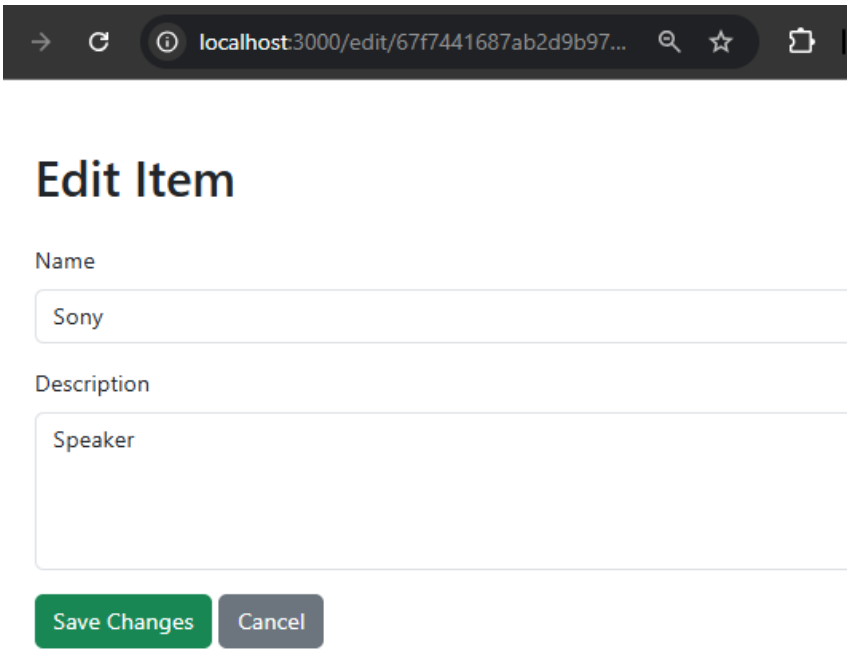

Figure 36: Edit Item Form

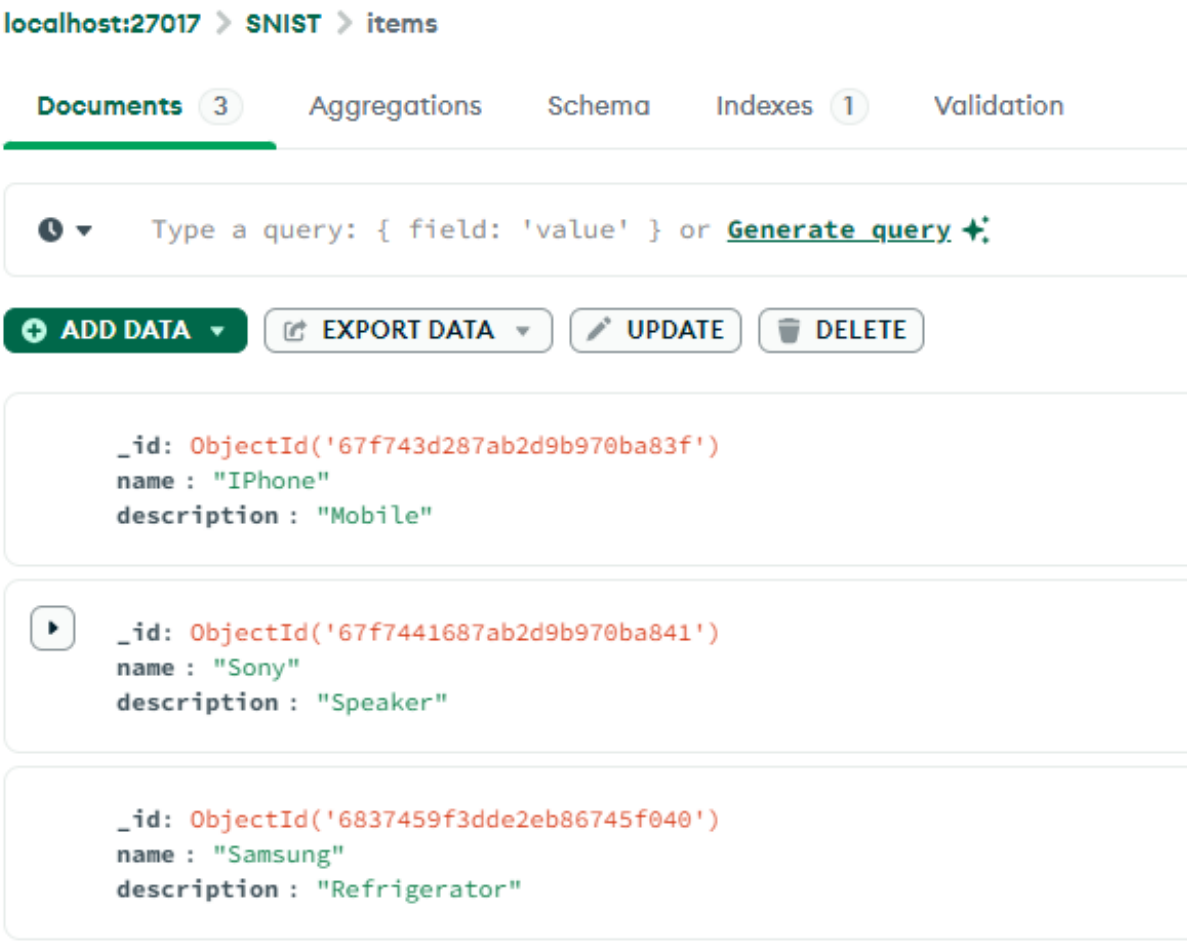

Figure 37: MongoDB Collection Data

15 Cycle 5 Program 2 Creating a File Upload Application with Express and MongoDB

Problem Statement

Develop a file upload application using Express.js and MongoDB. The application should allow user to upload files, store them in MongoDB, and retrieve them later. Implement file handling techniques to ensure secure and efficient file uploads and storage. Utilize MongoDB to store file metadata and references.

Setup Instructions

1. **Create a new project directory: mkdir file-upload-api**
2. **Navigate to the directory: cd file-upload-api**
3. **Initialize a new Node.js project: npm init -y**
4. **Install required packages: npm install express mongoose multer dotenv cors**
5. **Create files and directories:**
   - Create a main application file: **main.js**
   - Create a directory for database models: **mkdir models**
   - Inside 'models', create a file schema: **models/file.js**
   - Create a directory for routes: **mkdir routes**
   - Inside 'routes', create the file route handler: **routes/fileRoutes.js**
   - Create a **.env** file in the root directory and add your MongoDB connection string and desired port (e.g., 'MONGO_URL=mongodb://localhost:27017/filedb' and 'PORT=3000').

File Structure

```
file-upload-api/
    |-- main.js
    |-- .env
    |-- models/
        |-- file.js
    |-- routes/
        |-- fileRoutes.js
    |-- package.json
    |-- package-lock.json
```

Backend Development

1. **HTTP Methods**
   - This API utilizes various HTTP methods for file management:
     - **POST /files/upload**: Handles file uploads.
     - **GET /files/allfiles**: Retrieves metadata for all uploaded files.
     - **GET /files/:id**: Retrieves the binary data of a specific file by its ID.
     - **GET /files/metadata/:id**: Retrieves only the metadata (excluding binary data) of a specific file by its ID.
   - Understanding these methods is crucial for developing RESTful APIs, where different methods are used to perform different operations on resources.
2. **Routing Logic**
   - The code demonstrates how to define routes in an Express application using ‘express.Router()‘. Routing is fundamental in backend development, as it determines how the server responds to different HTTP requests at various endpoints.
3. **File Storage and Handling**
   - **Multer** is used to process ‘multipart/form-data‘ for file uploads. It’s configured to use ‘memoryStorage()‘ to store files as Buffers in memory temporarily before saving them to MongoDB.
   - A ‘fileFilter‘ is implemented to restrict uploads to specific image types (`jpg`, `jpeg`, `png`, `gif`) and a file size limit of 5MB is enforced.
4. **Database Integration**
   - **Mongoose** is used to connect to a MongoDB database and interact with file data. The ‘fileSchema‘ defines the structure for storing file information, including the binary data as a Buffer.
   - Files are saved to MongoDB, and retrieved either as full binary data or as metadata only, depending on the endpoint.
5. **Error Handling**
   - Each route includes ‘try-catch‘ blocks to gracefully handle potential errors during file operations (e.g., database connection issues, file not found, upload errors). Appropriate HTTP status codes and error messages are sent back to the client.
6. **Status Codes**
   - The use of HTTP status codes (200 for success, 201 for resource creation, 404 for not found, 500 for internal server error) is important in communicating the outcome of a request to the client. Properly setting status codes is essential for building a reliable API.

Code

main.js

```
const express = require('express');
const mongoose = require('mongoose');
const dotenv = require('dotenv');
const cors = require('cors');

dotenv.config(); // Load environment variables

const app = express();

app.use(cors());
app.use(express.json());

mongoose.connect(process.env.MONGO_URL)
    .then(() => console.log("Connected to MongoDB"))
    .catch((err) => console.log("Error connecting to MongoDB:", err));
app.use('/files',require('./routes/fileRoutes'));
app.listen(process.env.PORT);
console.log(`Server is running on port ${process.env.PORT}`);
```

models/file.js

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

const fileSchema = new mongoose.Schema({
    filename: String,
    originalname: String,
    contentType: String,
    data: Buffer,
    size: Number,
});
// Create a model from the schema
module.exports = mongoose.model('File', fileSchema);
```

routes/fileRoutes.js

```
const express = require('express');
const multer = require('multer'); //multer is a middleware for
   handling multipart/form-data, which is used for uploading files

const router = express.Router();
const File = require('../models/file'); // Import the File model

//use memory storage for multer
const storage = multer.memoryStorage();
const upload = multer({ storage, limits: { fileSize: 5 * 1024 * 1024
   }, fileFilter : (req, file,cb)=>{
    const allowedTypes = ['image/jpg','image/jpeg', 'image/png',
       'image/gif'];
    if(allowedTypes.includes(file.mimetype)) {
        cb(null, true);
    }
    else{
```

```
15          cb(null,false)
16      }
17  } }); // Limit file size to 5MB and allow only specific file types
        like jpeg, png, gif
18
19  // Route to handle file upload
20  router.post('/upload', upload.single('file'), async (req, res) => {
21      try{const file = new File({
22          filename: req.file.filename ,
23          originalname: req.file.originalname,
24          contentType: req.file.mimetype,
25          data: req.file.buffer,
26          size: req.file.size
27      });
28      await file.save();
29      res.status(201).json({ message: 'File uploaded successfully',
            fileId: file._id });
30      }
31      catch (error) {
32          console.error('Error uploading file:', error);
33          res.status(500).json({ message: 'Error uploading file' });
34      }
35  });
36  // Route to get all files
37  router.get('/allfiles', async (req, res) => {
38      try {
39          const files = await File.find().select('-data'); // Exclude
                the file data from the response
40          if (!files || files.length === 0) {
41              return res.status(404).json({ message: 'No files found' });
42          }
43          res.status(200).json(files);
44      } catch (error) {
45          console.error('Error retrieving files:', error);
46          res.status(500).json({ message: 'Error retrieving files' });
47      }
48  });
49
50  // Route to handle file retrieval by ID
51  router.get('/:id', async (req, res) => {
52      try {
53          const file = await File.findById(req.params.id);
54          if (!file) {
55              return res.status(404).json({ message: 'File not found' });
56          }
57          res.set('Content-Type', file.contentType);
58          res.set('Content-Disposition', `attachment;
                filename="${file.originalname}"`);
59          res.send(file.data);
60      } catch (error) {
61          console.error('Error retrieving file:', error);
62          res.status(500).json({ message: 'Error retrieving file' });
63      }
64  });
65  // Route to get all files
66
67
68  //Get metadata of a file
```

```
69 router.get('/metadata/:id', async (req, res) => {
70     try {
71         const file = await
               File.findById(req.params.id).select('-data'); // Exclude
               the file data from the response
72         if (!file) {
73             return res.status(404).json({ message: 'File not found' });
74         }
75         res.status(200).json(file);
76     } catch (error) {
77         console.error('Error retrieving file metadata:', error);
78         res.status(500).json({ message: 'Error retrieving file
               metadata' });
79     }
80 });
81
82 module.exports = router;
```

Testing and Debugging

- **Using Postman:**
  1. **Open Postman:** Launch the Postman application.
  2. **Start the server:** In your terminal, navigate to the 'file-upload-api' directory and run: **node main.js**
  3. **Upload a file (POST):**
     - Set method to **POST**.
     - URL: **http://localhost:your_port/files/upload**
     - In the 'Body' tab, select 'form-data'.
     - Add a key named 'file', select 'File' from the dropdown, and choose an image file from your computer.
     - Send the request and verify a 201 status with a success message and 'fileId'.
  4. **Get all files (GET):**
     - Set method to **GET**.
     - URL: **http://localhost:your_port/files/allfiles**
     - Send the request and verify a 200 status with a list of file metadata.
  5. **Get file by ID (GET):**
     - Set method to **GET**.
     - URL: **http://localhost:your_port/files/{fileId}** (replace 'fileId' with an ID from the upload response or 'allfiles' list).
     - Send the request and verify the file is downloaded or displayed.
  6. **Get file metadata by ID (GET):**
     - Set method to **GET**.
     - URL: **http://localhost:your_port/files/metadata/{fileId}**
     - Send the request and verify a 200 status with only the file's metadata.
  7. Test error cases for invalid file types, oversized files, or non-existent IDs.

—

Output Screens

Run the Command : **node main.js**
Output on command-line:

```
Connected to MongoDB
Server is running on port 3000
```

(Note: The port number will depend on what you've set in your '.env' file.)

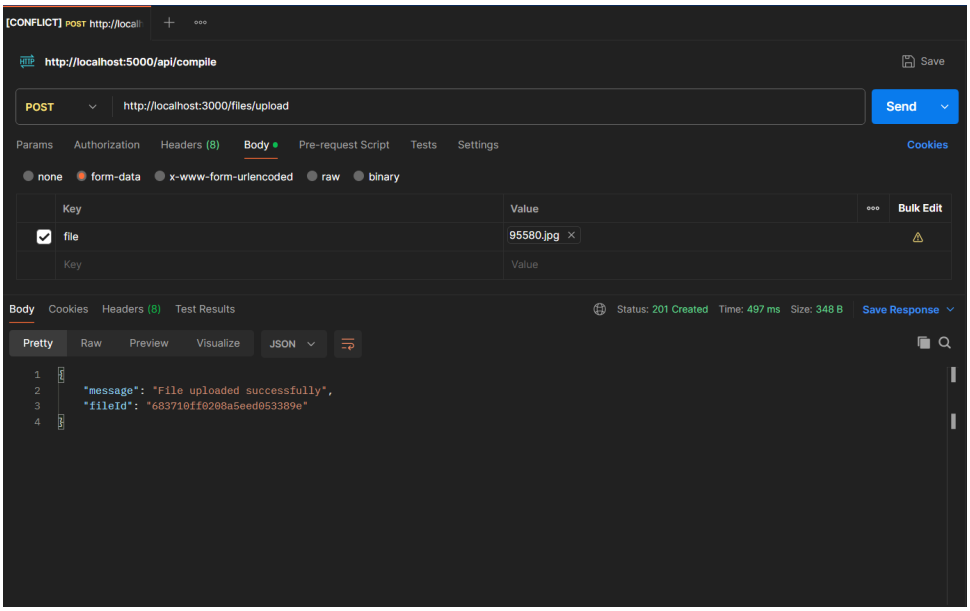

Figure 38: Upload file End point: Output

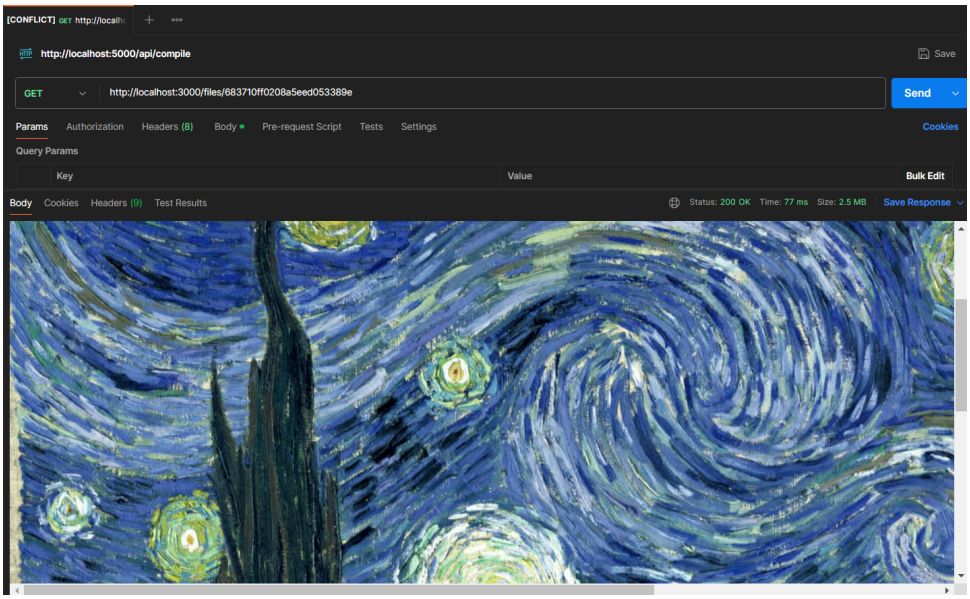

Figure 39: File view by ID: Output

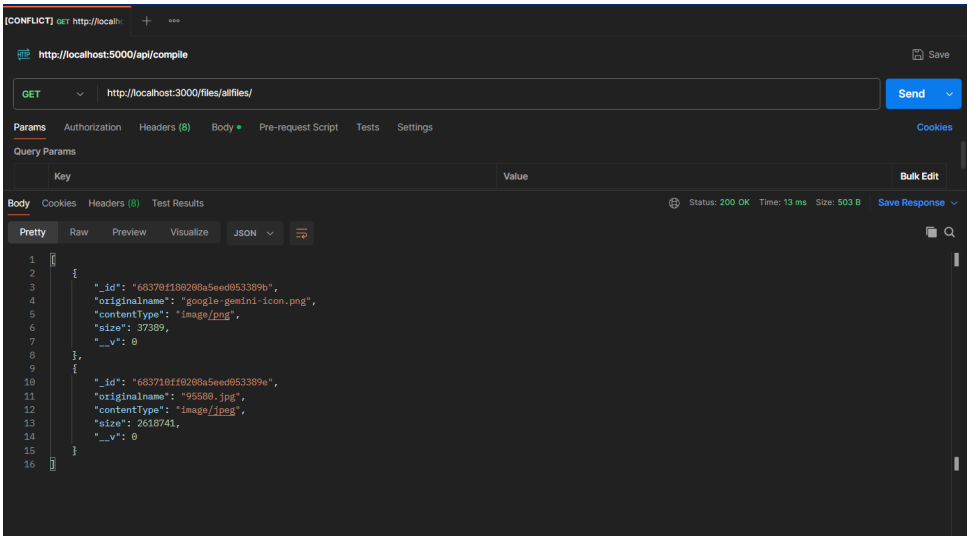

Figure 40: All Files End point: Output

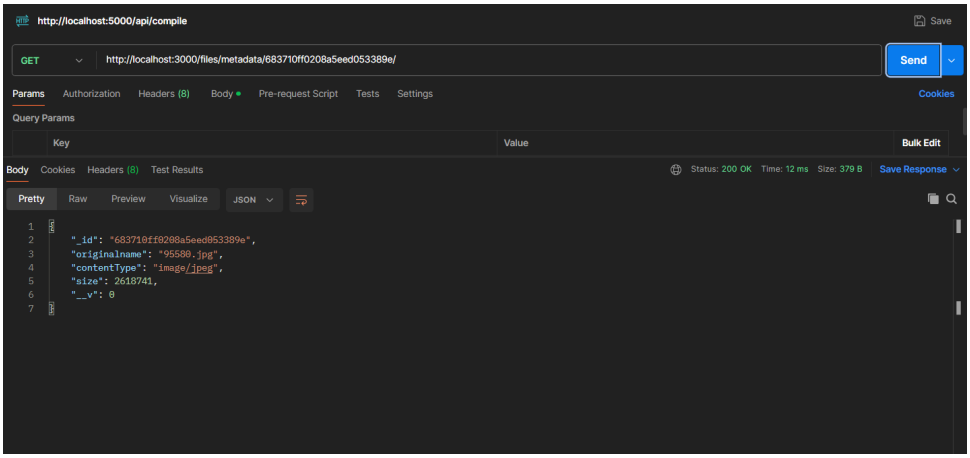

Figure 41: File Metadata Output: Output

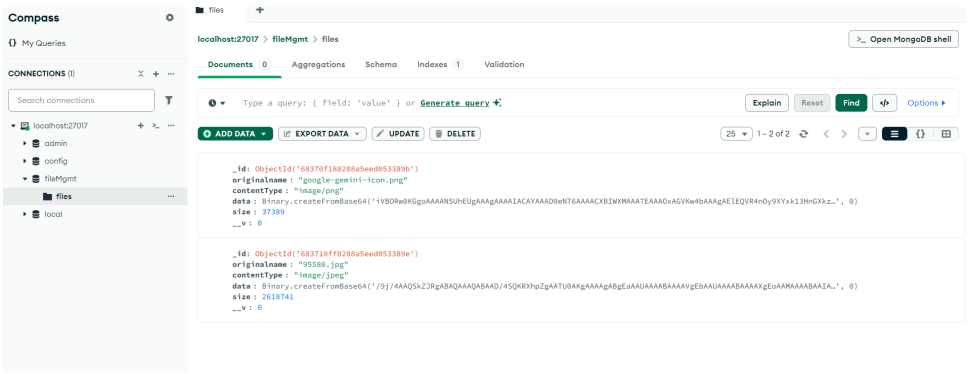

Figure 42: MongoDB Server after Uploads: Output

16 Cycle 6 Program 1 Building a Dynamic ReactJS Application with State Management

Problem Statement

Create a dynamic ReactJS application that manages state using hooks. The application should display a list of items and allow users to add, remove, and update items. Implement state management techniques to keep the UI in sync with data changes. Utilize hooks like useState and useEffect to handle state updates and side effects.

Technologies Used

- **ReactJS**: A JavaScript library for building user interfaces.
- **Node.js** (for npm/yarn): A JavaScript runtime environment for managing project dependencies.
- **npm** or **yarn**: Package managers for Node.js.

Setup Instructions

1. **Create a new React project:**
   npx create-react-app@latest item-manager-app
   (or **yarn create react-app item-manager-app**)
2. **Navigate to the project directory: cd item-manager-app**
3. **Open the project in your code editor** (e.g., VS Code).
4. **Replace the content of 'src/App.js' with the provided 'app.js' code.**
5. **Start the development server: npm start** (or **yarn start**)

File Structure

```
item-manager-app/
    |-- public/
    |   `-- index.html
    |-- src/
    |   `-- App.js
    |   `-- index.js
    |   `-- ... (other create-react-app generated files)
    |-- package.json
    |-- package-lock.json
    `-- node_modules/
```

Frontend Development

1. **React Components and JSX**
   - The application is built as a single functional React component (‘App‘).
   - **JSX** is used to define the UI structure, combining HTML-like syntax with JavaScript.
2. **State Management with ‘useState‘ Hook**
   - ‘useState‘ is used to declare and manage the application’s state:
     - ‘items‘: An array to store the list of strings (items).
     - ‘input‘: A string to store the current value of the input field.
     - ‘editIndex‘: A number (or ‘null‘) to keep track of the index of the item being edited.
   - State update functions (‘setItems‘, ‘setInput‘, ‘setEditIndex‘) are used to trigger re-renders and update the UI.
3. **Side Effects with ‘useEffect‘ Hook**
   - The ‘useEffect‘ hook is used to perform side effects. In this application, it logs the ‘items‘ array to the console whenever it changes, demonstrating how to react to state updates.
4. **Event Handling**
   - Functions like ‘addItem‘, ‘editItem‘, and ‘deleteItem‘ are defined to handle user interactions (form submission, button clicks).
   - These functions update the component’s state, leading to a re-render of the UI to reflect the changes.
5. **Conditional Rendering and List Rendering**
   - Conditional rendering

     `(‘items.length === 0 && <p>No items yet.</p>‘)`

     is used to display a message when the item list is empty.
   - The ‘map‘ method is used to iterate over the ‘items‘ array and render a list of ‘li‘ elements, each representing an item with ”Edit” and ”Delete” buttons. A ‘key‘ prop is used for efficient list rendering.

Code

app.js

```
import React, { useState, useEffect } from "react";

function App() {
  //State for items
  const [items, setItems] = useState([]);
  const [input,setInput] = useState('');
```

```
  const [editIndex,setEditIndex] = useState(null);

  //Effect to log changes in items
  useEffect(() => {
   console.log("Items changed:", items);
  }, [items]);

  //Add or Update item
  const addItem = (e) => {
    e.preventDefault();//Prevent default form submission
    if(input.trim() === '') return; //Return if input is empty
    if(editIndex !== null){
      const updatedItems = [...items];
      updatedItems[editIndex] = input;
      setItems(updatedItems);
      setEditIndex(null);
    }else{
      setItems([...items, input]);
    }
    setInput('');
  };

  //Edit item
  const editItem = (index) => {
    setInput(items[index]);
    setEditIndex(index);
  };

  //Delete item
  const deleteItem = (index) => {
      const updatedItems = [...items];
      updatedItems.splice(index, 1);
      setItems(updatedItems);
   };

   return (
      <div style={{padding:"20px"}}>
         <h1>Item Manager</h1>
         <form onSubmit={addItem}>
         <input
         type="text"
         value={input}
         onChange={(e) => setInput(e.target.value)}
         placeholder="Enter item"
         />
         <button type="submit">{editIndex !== null ? 'Update' :
   'Add'}</button>
         </form>

         <ul>
            {items.length === 0 && <p>No items yet.</p>}
            {items.map((item, index) => (
               <li key={index}>
                  {item}
                  {'  '}
                  <button onClick={() => editItem(index)}>Edit</button>
                  {'  '}
                  <button onClick={() =>
```

```
    deleteItem(index)}>Delete</button>
                </li>
              ))}
            </ul>
        </div>
    );
}

export default App;
```

Testing and Debugging

- **Accessing the Application:**
  1. Ensure your development server is running (‘npm start‘ or ‘yarn start‘).
  2. Open your web browser and navigate to **http://localhost:3000**.
- **Functionality Testing:**
  1. **Add Item:** Type text into the input field and click the ”Add” button. Verify that the item appears in the list.
  2. **Edit Item:** Click the ”Edit” button next to an existing item. The item’s text should populate the input field, and the button text should change to ”Update”. Modify the text and click ”Update”. Verify the item is updated in the list.
  3. **Delete Item:** Click the ”Delete” button next to an item. Verify that the item is removed from the list.
  4. **Empty Input:** Try clicking ”Add” with an empty input field. Verify that no empty item is added to the list.
  5. **Initial State:** Reload the page. Verify that ”No items yet.” is displayed until you add the first item.
- **Debugging:**
  - Use your browser’s developer tools (F12) to inspect the console for ‘console.log‘ messages (e.g., ”Items changed:”) and any potential React warnings or errors.
  - Use the React Developer Tools browser extension to inspect the component state (‘items‘, ‘input‘, ‘editIndex‘) in real-time.
  - Check the Network tab in developer tools for any unexpected requests (though this application is purely client-side).

Output Screens

Run the Command : **npm start** or **yarn start**
Output on command-line:

```
Compiled successfully!

You can now view item-manager-app in the browser.
```

```
  Local:            http://localhost:3000
  On Your Network:  http://192.168.X.X:3000

Note that the development build is not optimized.
To create a production build, use npm run build.
```

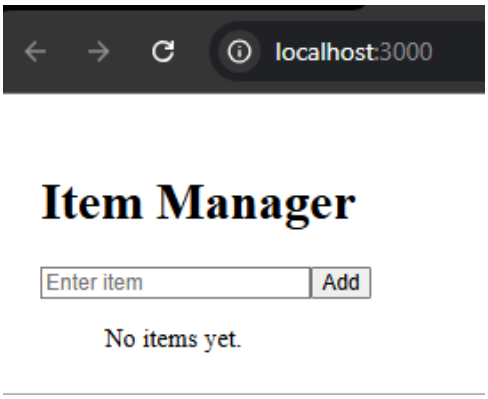

Figure 43: Initial Application State

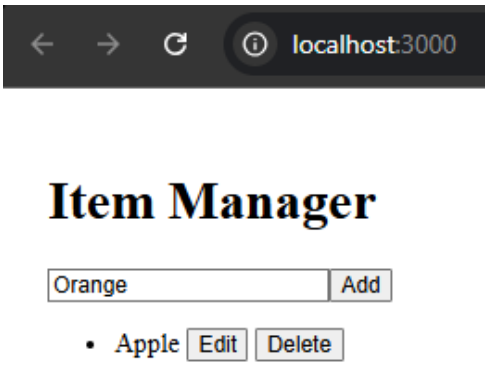

Figure 44: Adding a New Item

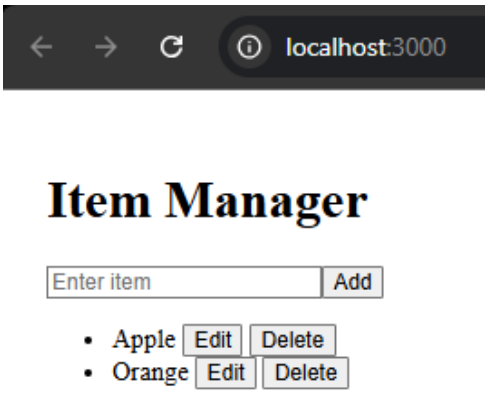

Figure 45: List of Items after Additions

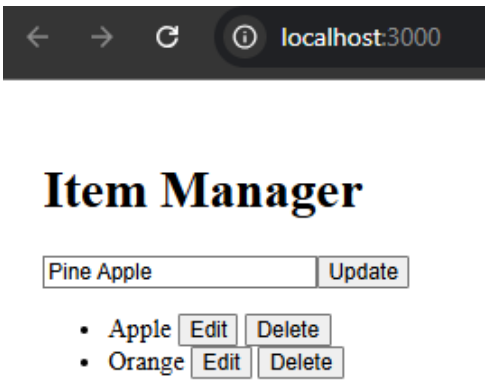

Figure 46: Editing an Existing Item

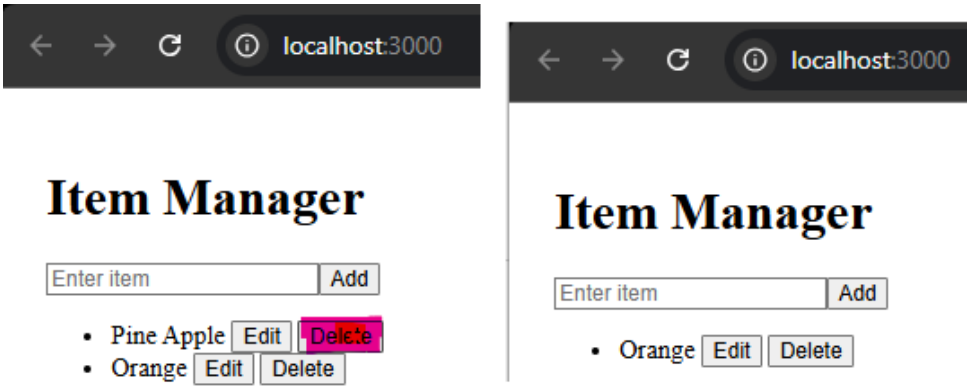

Figure 47: Deleting an Item

17 Cycle 6 Program 2 Developing a Data-Driven ReactJS Application with API Fetching

Problem Statement

Build a data-driven ReactJS application that fetches data from an API and renders it on the UI. The application should display a list of data items retrieved from the API and allow users to filter and search for items. Implement data fetching techniques using the fetch API or libraries like Axios. Utilize state management to store and update data from the API.

Technologies Used

- **ReactJS**: A JavaScript library for building user interfaces.
- **Node.js** (for npm/yarn): A JavaScript runtime environment for managing project dependencies.
- **npm** or **yarn**: Package managers for Node.js.
- **Fetch API**: Built-in browser API for making HTTP requests.
- **JSONPlaceholder API**: A free fake API for testing and prototyping.

Setup Instructions

1. **Create a new React project: npx create-react-app data-driven-app** (or **yarn create react-app data-driven-app**)
2. **Navigate to the project directory: cd data-driven-app**
3. **Open the project in your code editor** (e.g., VS Code).
4. **Replace the content of 'src/App.js' with the provided 'app.js' code.**
5. **Start the development server: npm start** (or **yarn start**)

File Structure

```
data-driven-app/
    |-- public/
    |   `-- index.html
    |-- src/
    |   `-- App.js
    |   `-- index.js
    |   `-- ... (other create-react-app generated files)
    |-- package.json
    |-- package-lock.json
    `-- node_modules/
```

Frontend Development

1. **React Components and JSX**
   - The application is built as a single functional React component (‘App‘).
   - **JSX** is used to define the UI structure, combining HTML-like syntax with JavaScript for rendering data.
2. **State Management with ‘useState‘ Hook**
   - ‘useState‘ is employed to manage various aspects of the application’s state:
     - ‘data‘: Stores the raw data fetched directly from the API.
     - ‘filterData‘: Stores the data that is currently displayed on the UI after filtering or searching.
     - ‘search‘: Stores the user’s current search query from the input field.
     - ‘loading‘: A boolean flag to indicate if data is currently being fetched (for displaying a loading message).
     - ‘error‘: Stores any error object that might occur during the API fetch process.
3. **Data Fetching with ‘useEffect‘ Hook**
   - The ‘useEffect‘ hook is used to perform the API call when the component mounts.
   - An ‘async‘ function ‘fetchData‘ is defined within ‘useEffect‘ to make an HTTP GET request to ‘https://jsonplaceholder.typicode.com/posts‘ using the ‘fetch‘ API.
   - It handles different states of the API call: ‘loading‘ (before fetch), ‘data‘ (on success), and ‘error‘ (on failure).
   - The empty dependency array ‘[]‘ ensures that the effect runs only once after the initial render.
4. **Search and Filtering Logic**
   - The ‘handleSearch‘ function is triggered whenever the user types into the search input.
   - It updates the ‘search‘ state and then filters the original ‘data‘ array based on whether the ‘item.title‘ includes the search query (case-insensitive).
   - The filtered results are then stored in ‘filterData‘, which controls what is displayed on the UI.
5. **Conditional Rendering**
   - The component conditionally renders different content based on the ‘loading‘, ‘error‘, and ‘filterData.length‘ states.
   - A ”Loading...” message is shown while data is being fetched.
   - An ”Error: [message]” is displayed if an error occurs.
   - If ‘filterData‘ is empty after a search, ”No data found” is displayed.
   - Otherwise, the ‘filterData‘ is mapped to a list of ‘¡li¿‘ elements, displaying the ‘title‘ and ‘body‘ of each post.

Code

app.js

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

export default function App() {
   const [data, setData] = useState([]); // to store the data
   const [filterData, setFilterData] = useState([]); // to store the
   filtered data
   const [search, setSearch] = useState(''); // to store the search
   query
   const [loading, setLoading] = useState(false); // to store the
   loading state
   const [error, setError] = useState(null); // to store the error

   //Fetch data from API
   useEffect(() => {
      const fetchData = async () => {
         setLoading(true);
         try {
            const response = await
   fetch('https://jsonplaceholder.typicode.com/posts');
            const data = await response.json();
            setData(data);
            setFilterData(data);
         } catch (error) {
            setError(error);
         } finally {
            setLoading(false);
         }
      };
      fetchData();
   }, []);

   //Handle Search
   const handleSearch = (query) => {
      setSearch(query);
      const filterData = data.filter((item) =>
         item.title.toLowerCase().includes(query.toLowerCase())
      );
      setFilterData(filterData);
   };

   if (loading) { return <div>Loading...</div>;    }
   if (error)    { return <div>Error: {error.message}</div>; }

   return (
      <div className="App">
         <h1>Data Driven ReactJS Application</h1>
         {/* Search Box */}
         <input
            type="text"
            placeholder="Search..."
            value={search}
            onChange={ (e) => handleSearch(e.target.value)}
         />
         {/* Data List */}
```

```
            {filterData.length > 0 ? (
            <ul>
               {filterData.map((item) => (
                  <li key={item.id}>
                     <strong>{item.title}</strong>
                     <p>{item.body}</p>
                  </li>
               ))}
            </ul>
         ) : (
            <p>No data found</p>
         )}
         </div>
   );
}
```

Testing and Debugging

- **Accessing the Application:**
  1. Ensure your development server is running (‘npm start‘ or ‘yarn start‘).
  2. Open your web browser and navigate to **http://localhost:3000**.
- **Functionality Testing:**
  1. **Initial Load:** Observe the ”Loading...” message briefly, then a list of posts should appear.
  2. **Search Functionality:** Type a keyword (e.g., ”sunt”, ”qui”, ”dolor”) into the search box. Verify that the list dynamically filters to show only matching posts.
  3. **No Results:** Type a non-existent keyword. Verify that ”No data found” is displayed. **Clear Search:** Delete the text in the search box. Verify that the full list of data reappears.
- **Debugging:**
  - Use your browser’s developer tools (F12) to inspect the console for any errors during API fetching or rendering.
  - Check the Network tab in developer tools to see the API request to ‘https://jsonplaceholder.ty and its response. Ensure the status code is 200 (OK).
  - Use the React Developer Tools browser extension to inspect the component’s state (‘data‘, ‘filterData‘, ‘search‘, ‘loading‘, ‘error‘) to understand how data flows and changes.
  - Temporarily introduce an invalid API URL in ‘fetch‘ to test the error handling (‘error‘ state).

—

Output Screens

Run the Command : **npm start** or **yarn start**
Output on command-line:

```
Compiled successfully!

You can now view data-driven-app in the browser.

  Local:            http://localhost:3000
  On Your Network:  http://192.168.X.X:3000

Note that the development build is not optimized.
To create a production build, use npm run build.
```

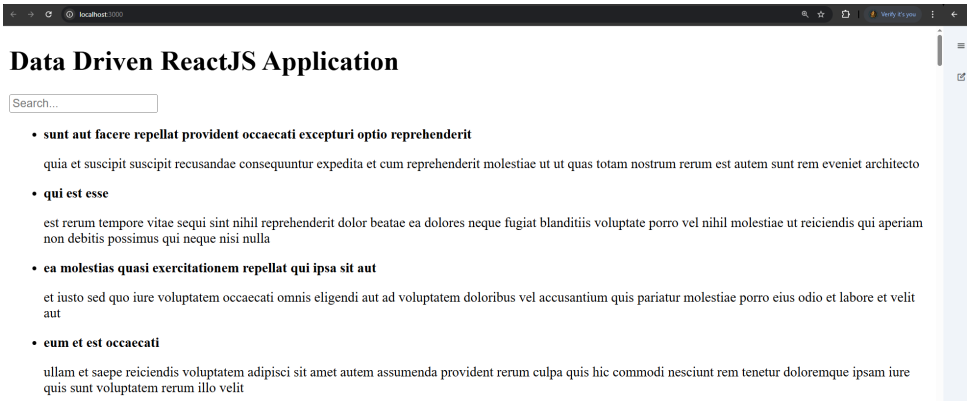

Figure 48: Initial Application Load with All Data

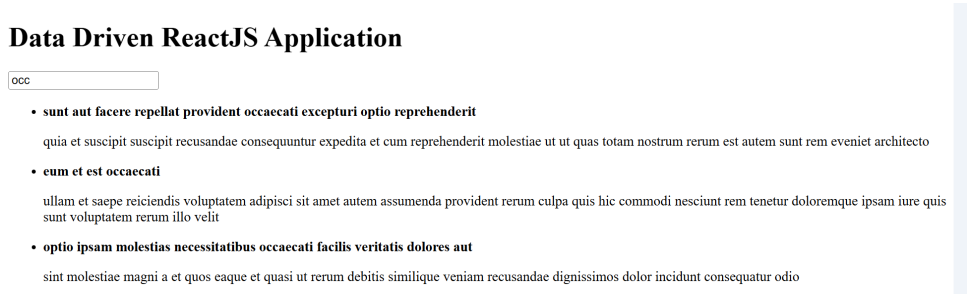

Figure 49: Filtered Data based on Search Query

Figure 50: No Data Found Message