SRM Institute of Science and Technology

Department of Computer Applications

Delhi – Meerut Road, Sikri Kalan, Ghaziabad, Uttar Pradesh – 201204

Circular – 2020-21

MCA GAI 3rd semester

Full Stack Development (PGI20D15J)

Lab Manual

Lab 1: Hello World Web Page

Title

Hello World Web Page using HTML

Aim

To create a basic web page that displays "Hello, World!" using HTML.

Procedure

1. Open a text editor (e.g., Notepad, VS Code, Sublime Text).
2. Type the HTML code provided in the "Source Code" section below.
3. Save the file with a `.html` extension (e.g., `hello.html`).
4. Open the saved `hello.html` file in any web browser (e.g., Chrome, Firefox, Edge) to view the output.

Source Code

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Hello World Page</title>
</head>
<body>
    <h1>Hello, World!</h1>
    <p>This is my first simple web page.</p>
</body>
</html>
```

Input

N/A (This is a static HTML page and does not require user input.)

Expected Output

A web page displayed in the browser with the title "Hello World Page" in the browser tab, and the text "Hello, World!" as a main heading, followed by "This is my first simple web page." on the page itself.

Lab 2: Create a website using HTML, CSS, and JavaScript

Title

Building a Simple Website with HTML, CSS, and JavaScript

Aim

To create a basic interactive website incorporating HTML for structure, CSS for styling, and JavaScript for dynamic behavior.

Procedure

1. Create three files in the same directory: `index.html`, `style.css`, and `script.js`.
2. Add the HTML structure to `index.html`, linking `style.css` and `script.js`.
3. Add CSS rules to `style.css` to style the HTML elements.
4. Add JavaScript code to `script.js` to add interactivity (e.g., changing text on a button click).
5. Open `index.html` in a web browser to view the website and test its interactivity.

Source 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>My Interactive Website</title>
    <link rel="stylesheet" href="style.css">
</head>
<body>
    <header>
        <h1>Welcome to My Website</h1>
    </header>
    <main>
        <p id="message">Click the button below!</p>
        <button id="myButton">Change Message</button>
    </main>
    <footer>
        <p>&copy; 2025 My Website</p>
    </footer>
    <script src="script.js"></script>
</body>
</html>
```

style.css

```
body {
    font-family: Arial, sans-serif;
    margin: 0;
    padding: 0;
    background-color: #f4f4f4;
    color: #333;
    display: flex;
    flex-direction: column;
    min-height: 100vh;
}
```

```
header {
    background-color: #333;
    color: #fff;
    padding: 1em 0;
    text-align: center;
}

main {
    flex: 1;
    padding: 20px;
    text-align: center;
}

#message {
    font-size: 1.2em;
    margin-bottom: 20px;
}

button {
    background-color: #007bff;
    color: white;
    padding: 10px 20px;
    border: none;
    border-radius: 5px;
    cursor: pointer;
    font-size: 1em;
    transition: background-color 0.3s ease;
}

button:hover {
    background-color: #0056b3;
}

footer {
    background-color: #333;
    color: #fff;
    text-align: center;
    padding: 1em 0;
    margin-top: auto;
}
```

script.js

```
document.addEventListener('DOMContentLoaded', () => {
    const myButton = document.getElementById('myButton');
    const messageElement = document.getElementById('message');

    myButton.addEventListener('click', () => {
        messageElement.textContent = 'Message changed! You clicked the button.';
    });
});
```

Input

User clicks the "Change Message" button.

Expected Output

Initially, the web page will display "Welcome to My Website" as a heading and "Click the button below!" as a paragraph, along with a button. After clicking the "Change Message" button, the text "Click the button below!" will change to "Message changed! You clicked the button.".

Lab 3: Build a Chat module using HTML, CSS, and JavaScript

Title

Building a Simple Chat Module with HTML, CSS, and JavaScript

Aim

To create a basic client-side chat interface using HTML for structure, CSS for styling, and JavaScript for sending and displaying messages within the browser.

Procedure

1. Create three files: `chat.html`, `chat.css`, and `chat.js` in the same directory.
2. Design the HTML structure for the chat interface in `chat.html`, including a message display area, an input field, and a send button. Link `chat.css` and `chat.js`.
3. Apply CSS styles in `chat.css` to make the chat module visually appealing and responsive.
4. Implement JavaScript logic in `chat.js` to handle sending messages (adding them to the display area) and clearing the input field.
5. Open `chat.html` in a web browser to test the chat functionality.

Source Code

chat.html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Simple Chat Module</title>
    <link rel="stylesheet" href="chat.css">
</head>
<body>
    <div class="chat-container">
        <div class="chat-header">
            <h2>Simple Chat</h2>
        </div>
        <div class="chat-messages" id="chatMessages">
            </div>
        <div class="chat-input">
            <input type="text" id="messageInput" placeholder="Type your
message...">
            <button id="sendMessageBtn">Send</button>
        </div>
    </div>
    <script src="chat.js"></script>
</body>
</html>
```

chat.css

```
body {
    font-family: 'Inter', sans-serif;
    display: flex;
    justify-content: center;
    align-items: center;
```

```
    min-height: 100vh;
    margin: 0;
    background-color: #f0f2f5;
    color: #333;
}

.chat-container {
    background-color: #fff;
    border-radius: 12px;
    box-shadow: 0 4px 20px rgba(0, 0, 0, 0.1);
    width: 90%;
    max-width: 500px;
    display: flex;
    flex-direction: column;
    overflow: hidden;
}

.chat-header {
    background-color: #007bff;
    color: white;
    padding: 15px 20px;
    text-align: center;
    font-size: 1.2em;
    border-top-left-radius: 12px;
    border-top-right-radius: 12px;
}

.chat-messages {
    flex-grow: 1;
    padding: 20px;
    overflow-y: auto;
    max-height: 400px; /* Limit height for scrollability */
    background-color: #e9ecef;
    border-bottom: 1px solid #dee2e6;
}

.message-bubble {
    background-color: #d1e7dd;
    padding: 10px 15px;
    border-radius: 8px;
    margin-bottom: 10px;
    max-width: 80%;
    word-wrap: break-word;
}

.chat-input {
    display: flex;
    padding: 15px 20px;
    background-color: #f8f9fa;
    border-bottom-left-radius: 12px;
    border-bottom-right-radius: 12px;
}

#messageInput {
    flex-grow: 1;
    padding: 10px 15px;
    border: 1px solid #ced4da;
    border-radius: 8px;
    font-size: 1em;
    margin-right: 10px;
}

#sendMessageBtn {
    background-color: #28a745;
    color: white;
    padding: 10px 20px;
    border: none;
```

```
    border-radius: 8px;
    cursor: pointer;
    font-size: 1em;
    transition: background-color 0.3s ease;
}

#sendMessageBtn:hover {
    background-color: #218838;
}
```

chat.js

```
document.addEventListener('DOMContentLoaded', () => {
    const messageInput = document.getElementById('messageInput');
    const sendMessageBtn = document.getElementById('sendMessageBtn');
    const chatMessages = document.getElementById('chatMessages');

    // Function to add a message to the chat display
    function addMessage(message) {
        const messageBubble = document.createElement('div');
        messageBubble.classList.add('message-bubble');
        messageBubble.textContent = message;
        chatMessages.appendChild(messageBubble);
        // Scroll to the bottom of the chat messages
        chatMessages.scrollTop = chatMessages.scrollHeight;
    }

    // Event listener for the Send button
    sendMessageBtn.addEventListener('click', () => {
        const message = messageInput.value.trim();
        if (message !== '') {
            addMessage(message);
            messageInput.value = ''; // Clear the input field
        }
    });

    // Event listener for pressing Enter key in the input field
    messageInput.addEventListener('keypress', (event) => {
        if (event.key === 'Enter') {
            sendMessageBtn.click(); // Simulate a click on the send button
        }
    });

    // Initial message
    addMessage('Welcome to the chat!');
});
```

Input

User types a message into the input field and presses Enter or clicks the "Send" button.

Expected Output

A chat interface with a header, a message display area, and an input field with a "Send" button. When a user types a message and sends it, the message will appear as a bubble in the chat display area, and the input field will clear.

Lab 4: Simple Node.js script that outputs "Hello, World!"

Title

Simple Node.js "Hello, World!" Script

Aim

To write and execute a basic Node.js script that prints "Hello, World!" to the console.

Procedure

1. **Install Node.js:** If you don't have Node.js installed, download and install it from the official website (nodejs.org).
2. Open a text editor (e.g., VS Code, Sublime Text, Notepad++).
3. Create a new file and save it as `hello.js`.
4. Type the JavaScript code provided in the "Source Code" section into `hello.js`.
5. Open your terminal or command prompt.
6. Navigate to the directory where you saved `hello.js` using the `cd` command.
7. Execute the script using the command: `node hello.js`.

Source Code

hello.js

```
// This is a simple Node.js script.
// It prints "Hello, World!" to the console.

console.log("Hello, World!");
```

Input

N/A (The script runs directly without requiring external input.)

Expected Output

When executed from the terminal, the following text will be displayed in the console:

```
Hello, World!
```

Lab 5: To-Do-List Application

Title

To-Do List Application with HTML, CSS, and JavaScript

Aim

To develop a web-based To-Do List application that allows users to add, mark as complete, and delete tasks.

Procedure

1. Create three files in the same directory: `todo.html`, `todo.css`, and `todo.js`.
2. Design the HTML structure for the To-Do list in `todo.html`, including an input field for new tasks, an "Add Task" button, and an unordered list to display tasks. Link the CSS and JavaScript files.
3. Style the To-Do list elements in `todo.css` to create a clean and user-friendly interface.
4. Implement JavaScript logic in `todo.js` to handle:
    - Adding new tasks to the list.
    - Toggling the completion status of a task (e.g., striking through the text).
    - Deleting tasks from the list.
5. Open `todo.html` in a web browser to interact with the To-Do list application.

Source Code

todo.html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>To-Do List App</title>
    <link rel="stylesheet" href="todo.css">
</head>
<body>
    <div class="todo-container">
        <h1>My To-Do List</h1>
        <div class="input-section">
            <input type="text" id="taskInput" placeholder="Add a new task...">
            <button id="addTaskBtn">Add Task</button>
        </div>
        <ul id="taskList">
            </ul>
    </div>
    <script src="todo.js"></script>
</body>
</html>
```

todo.css

```
body {
    font-family: 'Inter', sans-serif;
    display: flex;
    justify-content: center;
    align-items: flex-start; /* Align to top instead of center vertically */
    min-height: 100vh;
    margin: 0;
    padding-top: 50px; /* Add some padding from the top */
    background-color: #f0f2f5;
```

```
    color: #333;
}

.todo-container {
    background-color: #fff;
    border-radius: 12px;
    box-shadow: 0 6px 25px rgba(0, 0, 0, 0.15);
    width: 90%;
    max-width: 500px;
    padding: 30px;
    box-sizing: border-box; /* Include padding in element's total width and
height */
}

h1 {
    text-align: center;
    color: #007bff;
    margin-bottom: 30px;
}

.input-section {
    display: flex;
    margin-bottom: 25px;
}

#taskInput {
    flex-grow: 1;
    padding: 12px 15px;
    border: 1px solid #ced4da;
    border-radius: 8px;
    font-size: 1em;
    margin-right: 10px;
}

#addTaskBtn {
    background-color: #28a745;
    color: white;
    padding: 12px 20px;
    border: none;
    border-radius: 8px;
    cursor: pointer;
    font-size: 1em;
    transition: background-color 0.3s ease, transform 0.2s ease;
}

#addTaskBtn:hover {
    background-color: #218838;
    transform: translateY(-2px);
}

#taskList {
    list-style: none;
    padding: 0;
    margin: 0;
}

#taskList li {
    background-color: #f8f9fa;
    padding: 15px;
    margin-bottom: 10px;
    border-radius: 8px;
    display: flex;
    align-items: center;
    justify-content: space-between;
    transition: background-color 0.3s ease, box-shadow 0.3s ease;
    border: 1px solid #e9ecef;
}
```

```
#taskList li:hover {
    background-color: #e2e6ea;
    box-shadow: 0 2px 10px rgba(0, 0, 0, 0.08);
}

#taskList li.completed {
    text-decoration: line-through;
    color: #6c757d;
    background-color: #e2e6ea;
}

#taskList li span {
    flex-grow: 1;
    cursor: pointer;
    padding-right: 10px; /* Space between text and button */
}

#taskList li .delete-btn {
    background-color: #dc3545;
    color: white;
    border: none;
    padding: 8px 12px;
    border-radius: 5px;
    cursor: pointer;
    font-size: 0.9em;
    transition: background-color 0.3s ease;
}

#taskList li .delete-btn:hover {
    background-color: #c82333;
}
```

todo.js

```
document.addEventListener('DOMContentLoaded', () => {
    const taskInput = document.getElementById('taskInput');
    const addTaskBtn = document.getElementById('addTaskBtn');
    const taskList = document.getElementById('taskList');

    // Function to add a new task
    function addTask() {
        const taskText = taskInput.value.trim();

        if (taskText === '') {
            // Optionally, show a message to the user instead of an alert
            console.log("Task cannot be empty!");
            return;
        }

        // Create new list item
        const listItem = document.createElement('li');

        // Create span for task text
        const taskSpan = document.createElement('span');
        taskSpan.textContent = taskText;
        taskSpan.addEventListener('click', () => {
            listItem.classList.toggle('completed'); // Toggle 'completed' class
        });

        // Create delete button
        const deleteButton = document.createElement('button');
        deleteButton.textContent = 'Delete';
        deleteButton.classList.add('delete-btn');
        deleteButton.addEventListener('click', () => {
            taskList.removeChild(listItem); // Remove the task item
        });
```

```
        // Append span and button to list item
        listItem.appendChild(taskSpan);
        listItem.appendChild(deleteButton);

        // Append list item to the task list
        taskList.appendChild(listItem);

        // Clear the input field
        taskInput.value = '';
    }

    // Event listener for Add Task button
    addTaskBtn.addEventListener('click', addTask);

    // Event listener for Enter key on input field
    taskInput.addEventListener('keypress', (event) => {
        if (event.key === 'Enter') {
            addTask();
        }
    });
});
```

Input

User types a task into the input field and clicks "Add Task" or presses Enter. User clicks on a task to mark it as complete/incomplete. User clicks the "Delete" button next to a task.

Expected Output

A web page displaying a "To-Do List App" with an input field and an "Add Task" button.

- When a task is added, it appears in the list below.
- Clicking on a task toggles a strikethrough effect, indicating completion.
- Clicking the "Delete" button removes the task from the list.

Lab 6: Write a program to create a voting application using Angular JS

Title

Simple Voting Application using AngularJS

Aim

To develop a basic voting application using AngularJS that allows users to vote for predefined options and see the updated vote counts.

Procedure

1. Create an `index.html` file.
2. Include the AngularJS library via a CDN in `index.html`.
3. Define an AngularJS module and controller within a `<script>` tag in `index.html` or in a separate `app.js` file (for simplicity, we'll keep it in `index.html` for this basic example).
4. Design the HTML structure with `ng-app` and `ng-controller` directives.
5. Use `ng-repeat` to display voting options and `ng-click` to handle vote increments.
6. Display the vote counts using data binding.
7. Open `index.html` in a web browser to interact with the voting application.

Source 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>AngularJS Voting App</title>
    <script
src="https://ajax.googleapis.com/ajax/libs/angularjs/1.8.2/angular.min.js"></scr
ipt>
    <style>
        body {
            font-family: 'Inter', sans-serif;
            display: flex;
            justify-content: center;
            align-items: center;
            min-height: 100vh;
            margin: 0;
            background-color: #f0f2f5;
            color: #333;
        }

        .voting-container {
            background-color: #fff;
            border-radius: 12px;
            box-shadow: 0 6px 25px rgba(0, 0, 0, 0.15);
            padding: 30px;
            width: 90%;
            max-width: 600px;
            text-align: center;
        }
```

```
        h1 {
            color: #007bff;
            margin-bottom: 30px;
        }

        .option {
            display: flex;
            justify-content: space-between;
            align-items: center;
            background-color: #e9ecef;
            padding: 15px 20px;
            margin-bottom: 15px;
            border-radius: 8px;
            border: 1px solid #dee2e6;
            transition: background-color 0.3s ease;
        }

        .option:hover {
            background-color: #d1e7dd;
        }

        .option-name {
            font-size: 1.1em;
            font-weight: bold;
        }

        .vote-count {
            font-size: 1.2em;
            color: #28a745;
            font-weight: bold;
            min-width: 40px; /* Ensure consistent width */
            text-align: right;
        }

        button {
            background-color: #007bff;
            color: white;
            padding: 10px 15px;
            border: none;
            border-radius: 8px;
            cursor: pointer;
            font-size: 0.9em;
            transition: background-color 0.3s ease, transform 0.2s ease;
        }

        button:hover {
            background-color: #0056b3;
            transform: translateY(-2px);
        }
    </style>
</head>
<body ng-app="votingApp">
    <div class="voting-container" ng-controller="VotingController">
        <h1>Vote for your Favorite Option</h1>
        <div class="option" ng-repeat="option in options">
            <span class="option-name">{{ option.name }}</span>
            <div>
                <span class="vote-count">{{ option.votes }}</span>
                <button ng-click="vote(option)">Vote</button>
            </div>
        </div>
    </div>

    <script>
        // Define the AngularJS module
        var app = angular.module('votingApp', []);
```

```
        // Define the controller
        app.controller('VotingController', function($scope) {
            // Initial voting options
            $scope.options = [
                { name: 'Option A', votes: 0 },
                { name: 'Option B', votes: 0 },
                { name: 'Option C', votes: 0 }
            ];

            // Function to increment votes
            $scope.vote = function(option) {
                option.votes++;
            };
        });
    </script>
</body>
</html>
```

Input

User clicks the "Vote" button next to an option.

Expected Output

A web page displaying a "Vote for your Favorite Option" heading. Below it, there will be a list of options (e.g., "Option A", "Option B", "Option C"), each with its current vote count (initially 0) and a "Vote" button. When a "Vote" button is clicked, the corresponding option's vote count will increment by one.

Lab 7: Create different routes for handling HTTP GET requests using Express

Title

Express.js HTTP GET Routes

Aim

To create a simple Node.js application using Express.js that defines and handles multiple HTTP GET requests for different routes.

Procedure

1. **Initialize Node.js Project:**
    - Create a new directory for your project (e.g., `express-routes-app`).
    - Navigate into the directory in your terminal.
    - Initialize a new Node.js project: `npm init -y`
2. **Install Express:**
    - Install the Express.js framework: `npm install express`
3. **Create Server File:**
    - Create a file named `app.js` (or `server.js`) in your project directory.
4. **Write Express Code:**
    - Add the JavaScript code for the Express application, defining routes and their handlers, as shown in the "Source Code" section.
5. **Start the Server:**
    - Run the application from your terminal: `node app.js`
6. **Test Routes:**
    - Open your web browser or use a tool like Postman/Insomnia.
    - Navigate to the defined routes (e.g., `http://localhost:3000/`, `http://localhost:3000/about`, `http://localhost:3000/contact`).

Source Code

app.js

```
// Import the Express.js library
const express = require('express');

// Create an Express application instance
const app = express();

// Define the port number for the server to listen on
const PORT = 3000;

// --- Route Definitions ---

// Route 1: Home page route
// Handles GET requests to the root URL '/'
app.get('/', (req, res) => {
    // Send a simple text response to the client
    res.send('<h1>Welcome to the Home Page!</h1><p>Navigate to /about or
/contact</p>');
});
```

```
// Route 2: About page route
// Handles GET requests to '/about'
app.get('/about', (req, res) => {
    // Send a different text response for the about page
    res.send('<h1>About Us</h1><p>We are a company dedicated to web
development.</p>');
});

// Route 3: Contact page route
// Handles GET requests to '/contact'
app.get('/contact', (req, res) => {
    // Send a JSON response for the contact page
    res.json({
        title: 'Contact Us',
        email: 'info@example.com',
        phone: '+1234567890'
    });
});

// Route 4: Dynamic route with a parameter
// Handles GET requests to '/users/:userId'
// The :userId part is a route parameter that can capture values
app.get('/users/:userId', (req, res) => {
    // Access the route parameter using req.params
    const userId = req.params.userId;
    res.send(`<h1>User Profile</h1><p>You requested data for User ID:
${userId}</p>`);
});

// --- Server Start ---

// Start the Express server and listen for incoming requests on the specified
port
app.listen(PORT, () => {
    console.log(`Server is running on http://localhost:${PORT}`);
    console.log('Try visiting:');
    console.log(`- http://localhost:${PORT}/`);
    console.log(`- http://localhost:${PORT}/about`);
    console.log(`- http://localhost:${PORT}/contact`);
    console.log(`- http://localhost:${PORT}/users/123`); // Example dynamic
route
});
```

Input

Access the following URLs in a web browser or API client:

- `http://localhost:3000/`
- `http://localhost:3000/about`
- `http://localhost:3000/contact`
- `http://localhost:3000/users/your_id_here` (replace `your_id_here` with any value, e.g., `123`, `john_doe`)

Expected Output

- **For `http://localhost:3000/`:**
- `<h1>Welcome to the Home Page!</h1><p>Navigate to /about or /contact</p>`

- **For `http://localhost:3000/about`:**
- `<h1>About Us</h1><p>We are a company dedicated to web development.</p>`

- **For `http://localhost:3000/contact`:**
-
```
{
    "title": "Contact Us",
    "email": "info@example.com",
    "phone": "+1234567890"
}
```


- **For `http://localhost:3000/users/123`:**
-
```
<h1>User Profile</h1><p>You requested data for User ID: 123</p>
```


(The `123` will change based on the `userId` provided in the URL)

Lab 8: Write middleware functions to log requests, handle errors, and parse request bodies.

Title

Express.js Middleware for Logging, Error Handling, and Body Parsing

Aim

To implement custom middleware functions in an Express.js application for logging incoming requests, handling errors gracefully, and parsing JSON request bodies.

Procedure

1. **Initialize Node.js Project:**
    - Create a new directory (e.g., `express-middleware-app`).
    - Navigate into the directory.
    - Initialize a new Node.js project: `npm init -y`
2. **Install Express:**
    - Install the Express.js framework: `npm install express`
3. **Create Server File:**
    - Create a file named `app.js` (or `server.js`) in your project directory.
4. **Write Express Code with Middleware:**
    - Add the JavaScript code for the Express application, including the custom middleware functions and their application using `app.use()`, as shown in the "Source Code" section.
5. **Start the Server:**
    - Run the application from your terminal: `node app.js`
6. **Test Middleware:**
    - **Logging:** Make any GET request (e.g., `http://localhost:3000/`). Observe the console output.
    - **Body Parsing:** Use an API client (like Postman or Insomnia) to send a POST request to `http://localhost:3000/data` with a JSON body.
    - **Error Handling:** Make a request to an undefined route (e.g., `http://localhost:3000/nonexistent`) or a route that intentionally throws an error (e.g., `http://localhost:3000/error-test`).

Source Code

app.js

```
// Import the Express.js library
const express = require('express');

// Create an Express application instance
const app = express();

// Define the port number for the server to listen on
const PORT = 3000;

// --- Custom Middleware Functions ---

// 1. Request Logger Middleware
// This middleware logs details of every incoming request to the console.
```

```
const requestLogger = (req, res, next) => {
    const timestamp = new Date().toISOString();
    console.log(`[${timestamp}] ${req.method} ${req.url}`);
    // Call next() to pass control to the next middleware function in the stack
    next();
};

// 2. Error Handling Middleware
// This middleware catches errors that occur during request processing.
// It must have four arguments: (err, req, res, next)
const errorHandler = (err, req, res, next) => {
    console.error(`Error caught by middleware: ${err.stack}`); // Log the error
stack for debugging
    // Send a 500 Internal Server Error response to the client
    res.status(500).send('Something broke! Please try again later.');
};

// --- Apply Middleware ---

// Apply the requestLogger middleware to all incoming requests.
// It will execute for every request before any route handlers.
app.use(requestLogger);

// Built-in Express middleware to parse JSON request bodies.
// This is crucial for handling data sent with POST/PUT requests with 'Content-
Type: application/json'.
app.use(express.json());

// --- Route Definitions ---

// Home route
app.get('/', (req, res) => {
    res.send('Welcome to the Middleware Demo App!');
});

// Route to test JSON body parsing
app.post('/data', (req, res) => {
    // req.body will contain the parsed JSON data thanks to express.json()
middleware
    if (req.body) {
        console.log('Received data:', req.body);
        res.json({ message: 'Data received successfully!', yourData: req.body
});
    } else {
        res.status(400).send('No data received in the request body.');
    }
});

// Route to intentionally throw an error for testing error handling middleware
app.get('/error-test', (req, res, next) => {
    // Simulate an error
    const error = new Error('This is a test error!');
    // Pass the error to the next middleware (which will be our errorHandler)
    next(error);
});

// --- Error Handling and 404 (Not Found) Middleware ---

// This middleware will be executed if no other route matches the request.
// It should be placed after all other routes.
app.use((req, res, next) => {
    res.status(404).send('Sorry, that page cannot be found!');
});

// Apply the error handling middleware.
// This should be the very last middleware in the chain.
app.use(errorHandler);
```

```
// --- Server Start ---

// Start the Express server
app.listen(PORT, () => {
    console.log(`Server is running on http://localhost:${PORT}`);
    console.log('Test routes:');
    console.log('- GET / (Check console for request log)');
    console.log('- POST /data (Send JSON body like {"name": "Test", "value":
123})');
    console.log('- GET /error-test (To trigger the error handler)');
    console.log('- GET /nonexistent (To trigger the 404 handler)');
});
```

Input

1. **For Logging:**
    - Open `http://localhost:3000/` in your browser.
2. **For Body Parsing:**
    - Use Postman/Insomnia.
    - Method: `POST`
    - URL: `http://localhost:3000/data`
    - Headers: `Content-Type: application/json`
    - Body (raw JSON):
    - ```
      {
          "productName": "Laptop",
          "price": 1200,
          "quantity": 1
      }
      ```
3. **For Error Handling:**
    - Open `http://localhost:3000/error-test` in your browser.
    - Open `http://localhost:3000/nonexistent` in your browser.

Expected Output

1. **For Logging:**
    - In your terminal where the Node.js server is running, you will see output similar to:
    - ```
      [2025-05-22TXX:XX:XX.XXXZ] GET /
      ```

      (Timestamp and method/URL will vary based on your request)

2. **For Body Parsing (`POST /data`):**
    - **Client Response:**
    - ```
      {
          "message": "Data received successfully!",
          "yourData": {
              "productName": "Laptop",
              "price": 1200,
              "quantity": 1
          }
      }
      ```
    - **Server Console:**
    - ```
      Received data: { productName: 'Laptop', price: 1200, quantity: 1 }
      ```

3. **For Error Handling (`GET /error-test`):**
    - **Client Response:**
    - `Something broke! Please try again later.`

    - **Server Console:**
    - `Error caught by middleware: Error: This is a test error!`
    - `    at C:\path\to\your\app.js:XX:XX (stack trace will follow)`

4. **For 404 Handling (`GET /nonexistent`):**
    - **Client Response:**
    - `Sorry, that page cannot be found!`

Lab 9: Write middleware functions to log requests, handle errors, and parse request bodies.

Title

Express.js Middleware for Logging, Error Handling, and Body Parsing (Duplicate)

Aim

This lab is a duplicate of Lab 8. The aim remains the same: to implement custom middleware functions in an Express.js application for logging incoming requests, handling errors gracefully, and parsing JSON request bodies.

Procedure

The procedure is identical to Lab 8.

1. **Initialize Node.js Project:**
    - Create a new directory (e.g., `express-middleware-app-2`).
    - Navigate into the directory.
    - Initialize a new Node.js project: `npm init -y`
2. **Install Express:**
    - Install the Express.js framework: `npm install express`
3. **Create Server File:**
    - Create a file named `app.js` (or `server.js`) in your project directory.
4. **Write Express Code with Middleware:**
    - Add the JavaScript code for the Express application, including the custom middleware functions and their application using `app.use()`, as shown in the "Source Code" section.
5. **Start the Server:**
    - Run the application from your terminal: `node app.js`
6. **Test Middleware:**
    - **Logging:** Make any GET request (e.g., `http://localhost:3000/`). Observe the console output.
    - **Body Parsing:** Use an API client (like Postman or Insomnia) to send a POST request to `http://localhost:3000/data` with a JSON body.
    - **Error Handling:** Make a request to an undefined route (e.g., `http://localhost:3000/nonexistent`) or a route that intentionally throws an error (e.g., `http://localhost:3000/error-test`).

Source Code

app.js

```
// Import the Express.js library
const express = require('express');

// Create an Express application instance
const app = express();

// Define the port number for the server to listen on
const PORT = 3000;
```

```
// --- Custom Middleware Functions ---

// 1. Request Logger Middleware
// This middleware logs details of every incoming request to the console.
const requestLogger = (req, res, next) => {
    const timestamp = new Date().toISOString();
    console.log(`[${timestamp}] ${req.method} ${req.url}`);
    // Call next() to pass control to the next middleware function in the stack
    next();
};

// 2. Error Handling Middleware
// This middleware catches errors that occur during request processing.
// It must have four arguments: (err, req, res, next)
const errorHandler = (err, req, res, next) => {
    console.error(`Error caught by middleware: ${err.stack}`); // Log the error
stack for debugging
    // Send a 500 Internal Server Error response to the client
    res.status(500).send('Something broke! Please try again later.');
};

// --- Apply Middleware ---

// Apply the requestLogger middleware to all incoming requests.
// It will execute for every request before any route handlers.
app.use(requestLogger);

// Built-in Express middleware to parse JSON request bodies.
// This is crucial for handling data sent with POST/PUT requests with 'Content-
Type: application/json'.
app.use(express.json());

// --- Route Definitions ---

// Home route
app.get('/', (req, res) => {
    res.send('Welcome to the Middleware Demo App (Duplicate Lab)!');
});

// Route to test JSON body parsing
app.post('/data', (req, res) => {
    // req.body will contain the parsed JSON data thanks to express.json()
middleware
    if (req.body) {
        console.log('Received data:', req.body);
        res.json({ message: 'Data received successfully!', yourData: req.body
});
    } else {
        res.status(400).send('No data received in the request body.');
    }
});

// Route to intentionally throw an error for testing error handling middleware
app.get('/error-test', (req, res, next) => {
    // Simulate an error
    const error = new Error('This is a test error from Duplicate Lab!');
    // Pass the error to the next middleware (which will be our errorHandler)
    next(error);
});

// --- Error Handling and 404 (Not Found) Middleware ---

// This middleware will be executed if no other route matches the request.
// It should be placed after all other routes.
app.use((req, res, next) => {
    res.status(404).send('Sorry, that page cannot be found in Duplicate Lab!');
});
```

```
// Apply the error handling middleware.
// This should be the very last middleware in the chain.
app.use(errorHandler);

// --- Server Start ---

// Start the Express server
app.listen(PORT, () => {
    console.log(`Server is running on http://localhost:${PORT}`);
    console.log('Test routes for Duplicate Lab:');
    console.log('- GET / (Check console for request log)');
    console.log('- POST /data (Send JSON body like {"name": "Test", "value":
123})');
    console.log('- GET /error-test (To trigger the error handler)');
    console.log('- GET /nonexistent (To trigger the 404 handler)');
});
```

Input

1. **For Logging:**
    - Open `http://localhost:3000/` in your browser.
2. **For Body Parsing:**
    - Use Postman/Insomnia.
    - Method: `POST`
    - URL: `http://localhost:3000/data`
    - Headers: `Content-Type: application/json`
    - Body (raw JSON):
    - ```
      {
          "item": "Book",
          "isbn": "978-3-16-148410-0"
      }
      ```
3. **For Error Handling:**
    - Open `http://localhost:3000/error-test` in your browser.
    - Open `http://localhost:3000/nonexistent` in your browser.

Expected Output

1. **For Logging:**
    - In your terminal where the Node.js server is running, you will see output similar to:
    - ```
      [2025-05-22TXX:XX:XX.XXXz] GET /
      ```

      (Timestamp and method/URL will vary based on your request)
2. **For Body Parsing (`POST /data`):**
    - **Client Response:**
    - ```
      {
          "message": "Data received successfully!",
          "yourData": {
              "item": "Book",
              "isbn": "978-3-16-148410-0"
          }
      }
      ```
    - **Server Console:**
      ```

- Received data: { item: 'Book', isbn: '978-3-16-148410-0' }

3. **For Error Handling (`GET /error-test`):**
   - **Client Response:**
   - Something broke! Please try again later.

   - **Server Console:**
   - Error caught by middleware: Error: This is a test error from Duplicate Lab!
   -     at C:\path\to\your\app.js:XX:XX (stack trace will follow)

4. **For 404 Handling (`GET /nonexistent`):**
   - **Client Response:**
   - Sorry, that page cannot be found in Duplicate Lab!

Lab 10: CRUD Operations

Title

Node.js Express CRUD API with In-Memory Data

Aim

To implement a RESTful API using Node.js and Express.js that supports Create, Read, Update, and Delete (CRUD) operations on a collection of in-memory data.

Procedure

1. **Initialize Node.js Project:**
    - Create a new directory (e.g., `express-crud-app`).
    - Navigate into the directory.
    - Initialize a new Node.js project: `npm init -y`
2. **Install Express:**
    - Install the Express.js framework: `npm install express`
3. **Create Server File:**
    - Create a file named `app.js` (or `server.js`) in your project directory.
4. **Write Express Code for CRUD:**
    - Add the JavaScript code for the Express application, defining routes for POST (Create), GET (Read), PUT (Update), and DELETE (Delete) operations, as shown in the "Source Code" section. Use `express.json()` middleware for parsing request bodies.
5. **Start the Server:**
    - Run the application from your terminal: `node app.js`
6. **Test CRUD Operations:**
    - Use an API client (like Postman or Insomnia) to send HTTP requests to the defined endpoints.

Source Code

app.js

```
// Import the Express.js library
const express = require('express');

// Create an Express application instance
const app = express();

// Define the port number for the server to listen on
const PORT = 3000;

// In-memory data store (simulating a database)
// We'll store 'items' with a unique ID and a name
let items = [
    { id: 1, name: 'Apple' },
    { id: 2, name: 'Banana' }
];
let nextId = 3; // To generate unique IDs for new items

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

```
// --- CRUD Routes ---

// 1. CREATE: Add a new item (POST /items)
app.post('/items', (req, res) => {
    const { name } = req.body; // Extract name from request body
    if (!name) {
        return res.status(400).json({ message: 'Item name is required.' });
    }
    const newItem = { id: nextId++, name }; // Create new item with unique ID
    items.push(newItem); // Add to our in-memory array
    res.status(201).json(newItem); // Respond with the created item and 201
Created status
});

// 2. READ All: Get all items (GET /items)
app.get('/items', (req, res) => {
    res.json(items); // Respond with the entire list of items
});

// 3. READ One: Get a single item by ID (GET /items/:id)
app.get('/items/:id', (req, res) => {
    const id = parseInt(req.params.id); // Get ID from URL parameter and convert
to integer
    const item = items.find(item => item.id === id); // Find item by ID

    if (!item) {
        return res.status(404).json({ message: 'Item not found.' }); // 404 Not
Found if item doesn't exist
    }
    res.json(item); // Respond with the found item
});

// 4. UPDATE: Update an existing item by ID (PUT /items/:id)
app.put('/items/:id', (req, res) => {
    const id = parseInt(req.params.id);
    const { name } = req.body;

    if (!name) {
        return res.status(400).json({ message: 'New item name is required for
update.' });
    }

    const itemIndex = items.findIndex(item => item.id === id); // Find index of
item

    if (itemIndex === -1) {
        return res.status(404).json({ message: 'Item not found for update.' });
    }

    items[itemIndex].name = name; // Update the item's name
    res.json(items[itemIndex]); // Respond with the updated item
});

// 5. DELETE: Delete an item by ID (DELETE /items/:id)
app.delete('/items/:id', (req, res) => {
    const id = parseInt(req.params.id);

    const initialLength = items.length;
    // Filter out the item to be deleted
    items = items.filter(item => item.id !== id);

    if (items.length === initialLength) {
        return res.status(404).json({ message: 'Item not found for deletion.'
}); // If length didn't change, item wasn't found
    }
    res.status(204).send(); // Respond with 204 No Content for successful
deletion
```

```
});

// --- Server Start ---

// Start the Express server
app.listen(PORT, () => {
    console.log(`CRUD API server is running on http://localhost:${PORT}`);
    console.log('Endpoints:');
    console.log('  POST   /items         - Create a new item (e.g., {"name":
"Orange"})');
    console.log('  GET    /items         - Get all items');
    console.log('  GET    /items/:id     - Get an item by ID (e.g., /items/1)');
    console.log('  PUT    /items/:id     - Update an item by ID (e.g., /items/1,
{"name": "Grape"})');
    console.log('  DELETE /items/:id     - Delete an item by ID (e.g.,
/items/1)');
});
```

Input

You will use an API client (like Postman, Insomnia, or `curl`) to send HTTP requests.

1. **Create (POST):**
    - Method: `POST`
    - URL: `http://localhost:3000/items`
    - Headers: `Content-Type: application/json`
    - Body (raw JSON):
    - `{`
    - `    "name": "Orange"`
    - `}`

2. **Read All (GET):**
    - Method: `GET`
    - URL: `http://localhost:3000/items`
3. **Read One (GET by ID):**
    - Method: `GET`
    - URL: `http://localhost:3000/items/1` (or `2`, or the ID of an item you just created)
4. **Update (PUT):**
    - Method: `PUT`
    - URL: `http://localhost:3000/items/1` (update item with ID 1)
    - Headers: `Content-Type: application/json`
    - Body (raw JSON):
    - `{`
    - `    "name": "Grape"`
    - `}`

5. **Delete (DELETE):**
    - Method: `DELETE`
    - URL: `http://localhost:3000/items/2` (delete item with ID 2)

Expected Output

1. **Create (POST /items):**
    - **Status:** `201 Created`
    - **Body:**

```
{
    "id": 3,
    "name": "Orange"
}
```

(ID will increment with each new item)

2. **Read All (GET /items):**
   - **Status:** `200 OK`
   - **Body:** (Example after creating "Orange" and before updating/deleting)

```
[
    { "id": 1, "name": "Apple" },
    { "id": 2, "name": "Banana" },
    { "id": 3, "name": "Orange" }
]
```

3. **Read One (GET /items/1):**
   - **Status:** `200 OK`
   - **Body:**

```
{
    "id": 1,
    "name": "Apple"
}
```

4. **Update (PUT /items/1):**
   - **Status:** `200 OK`
   - **Body:**

```
{
    "id": 1,
    "name": "Grape"
}
```

5. **Delete (DELETE /items/2):**
   - **Status:** `204 No Content`
   - **Body:** (Empty)

Lab 11: Query Language

Title

Database Query Language Concepts (SQL Example)

Aim

To understand and demonstrate fundamental concepts of database query languages, specifically using SQL (Structured Query Language) for data retrieval, filtering, and ordering.

Procedure

1. **Choose a Database System:** For this lab, we'll assume a relational database like SQLite, MySQL, or PostgreSQL. You'll need a way to run SQL queries (e.g., a database client, a command-line interface, or an online SQL sandbox).
2. **Create a Sample Database:** Create a simple database and a table (e.g., `Students`) with some sample data.
3. **Execute SQL Queries:** Write and execute the SQL queries provided in the "Source Code" section.
4. **Observe Results:** Analyze the output of each query to understand its effect on the data.

Source Code

SQL Queries (Example for a `Students` table)

Assume a table named `Students` with the following structure and data:

Table: `Students` | `StudentID` | `FirstName` | `LastName` | `Age` | `Major` | `GPA` | | :---------- | :---------- | :---------- | :---- | :---------- | :---- | | 1 | Alice | Smith | 20 | Computer Science | 3.8 | | 2 | Bob | Johnson | 22 | Engineering | 3.5 | | 3 | Charlie | Brown | 21 | Computer Science | 3.9 | | 4 | Diana | Miller | 20 | Arts | 3.2 | | 5 | Eve | Davis | 23 | Engineering | 3.7 |

1. Create Table and Insert Data (Setup)

```
-- Create the Students table
CREATE TABLE Students (
    StudentID INT PRIMARY KEY,
    FirstName VARCHAR(50),
    LastName VARCHAR(50),
    Age INT,
    Major VARCHAR(100),
    GPA DECIMAL(3, 2)
);

-- Insert sample data into the Students table
INSERT INTO Students (StudentID, FirstName, LastName, Age, Major, GPA) VALUES
(1, 'Alice', 'Smith', 20, 'Computer Science', 3.8),
(2, 'Bob', 'Johnson', 22, 'Engineering', 3.5),
(3, 'Charlie', 'Brown', 21, 'Computer Science', 3.9),
(4, 'Diana', 'Miller', 20, 'Arts', 3.2),
(5, 'Eve', 'Davis', 23, 'Engineering', 3.7);
```

2. Select All Data

```
-- Select all columns and all rows from the Students table
SELECT * FROM Students;
```

3. Select Specific Columns

```
-- Select only FirstName, LastName, and Major columns
SELECT FirstName, LastName, Major FROM Students;
```

4. Filter Data (WHERE clause)

```
-- Select students majoring in 'Computer Science'
SELECT * FROM Students WHERE Major = 'Computer Science';
```

5. Filter Data with Multiple Conditions (AND / OR)

```
-- Select students older than 20 AND majoring in 'Engineering'
SELECT * FROM Students WHERE Age > 20 AND Major = 'Engineering';
```

6. Order Data (ORDER BY clause)

```
-- Select all students, ordered by GPA in descending order
SELECT * FROM Students ORDER BY GPA DESC;
```

7. Aggregate Functions (COUNT, AVG, SUM, MIN, MAX)

```
-- Count the total number of students
SELECT COUNT(*) AS TotalStudents FROM Students;

-- Calculate the average GPA of all students
SELECT AVG(GPA) AS AverageGPA FROM Students;
```

8. Group Data (GROUP BY clause)

```
-- Count students per major
SELECT Major, COUNT(*) AS NumberOfStudents FROM Students GROUP BY Major;
```

Input

The SQL queries are executed against a database. No direct user input during query execution, but the data in the `Students` table serves as input for the queries.

Expected Output

1. **`SELECT * FROM Students;`** | `StudentID` | `FirstName` | `LastName` | `Age` | `Major` | `GPA` | | :---------- | :---------- | :--------- | :---- | :--------------- | :---- | | 1 | Alice | Smith | 20 | Computer Science | 3.8 | | 2 | Bob | Johnson | 22 | Engineering | 3.5 | | 3 | Charlie | Brown | 21 | Computer Science | 3.9 | | 4 | Diana | Miller | 20 | Arts | 3.2 | | 5 | Eve | Davis | 23 | Engineering | 3.7 |

2. `SELECT FirstName, LastName, Major FROM Students;`

FirstName	LastName	Major
Alice	Smith	Computer Science
Bob	Johnson	Engineering
Charlie	Brown	Computer Science
Diana	Miller	Arts
Eve	Davis	Engineering

3. `SELECT * FROM Students WHERE Major = 'Computer Science';`

StudentID	FirstName	LastName	Age	Major	GPA
1	Alice	Smith	20	Computer Science	3.8
3	Charlie	Brown	21	Computer Science	3.9

4. `SELECT * FROM Students WHERE Age > 20 AND Major = 'Engineering';`

StudentID	FirstName	LastName	Age	Major	GPA
2	Bob	Johnson	22	Engineering	3.5
5	Eve	Davis	23	Engineering	3.7

5. `SELECT * FROM Students ORDER BY GPA DESC;`

StudentID	FirstName	LastName	Age	Major	GPA
3	Charlie	Brown	21	Computer Science	3.9
1	Alice	Smith	20	Computer Science	3.8
5	Eve	Davis	23	Engineering	3.7
2	Bob	Johnson	22	Engineering	3.5
4	Diana	Miller	20	Arts	3.2

6. `SELECT COUNT(*) AS TotalStudents FROM Students;`

TotalStudents
5

7. `SELECT AVG(GPA) AS AverageGPA FROM Students;`

AverageGPA
3.62

8. `SELECT Major, COUNT(*) AS NumberOfStudents FROM Students GROUP BY Major;`

Major	NumberOfStudents
Arts	1
Computer Science	2
Engineering	2

Lab 12: Writing a REST API " Exposing the MongoDB database to the application

Title

REST API with Node.js, Express, and MongoDB

Aim

To build a RESTful API using Node.js and Express.js that interacts with a MongoDB database, allowing for CRUD operations on a collection of resources (e.g., `products`).

Procedure

1. **Install MongoDB:** Ensure you have MongoDB installed and running on your system, or use a cloud-based MongoDB Atlas instance.
2. **Initialize Node.js Project:**
    - Create a new directory (e.g., `mongo-api-app`).
    - Navigate into the directory.
    - Initialize a new Node.js project: `npm init -y`
3. **Install Dependencies:**
    - Install Express and Mongoose (an ODM for MongoDB): `npm install express mongoose`
4. **Create Server File:**
    - Create a file named `app.js` (or `server.js`) in your project directory.
5. **Write API Code:**
    - Add the JavaScript code for connecting to MongoDB, defining a Mongoose schema and model, and implementing CRUD routes using Express, as shown in the "Source Code" section.
6. **Start the Server:**
    - Run the application from your terminal: `node app.js`
7. **Test API Endpoints:**
    - Use an API client (like Postman or Insomnia) to send HTTP requests to the defined endpoints.

Source Code

app.js

```
// Import necessary modules
const express = require('express');
const mongoose = require('mongoose');

// Create an Express application instance
const app = express();

// Define the port number for the server to listen on
const PORT = 3000;

// MongoDB connection URI
// IMPORTANT: Replace 'your_mongodb_connection_string' with your actual MongoDB
URI.
// For local MongoDB, it might be 'mongodb://localhost:27017/mydatabase'
// For MongoDB Atlas, it will be a longer string provided by Atlas.
const MONGODB_URI = 'mongodb://localhost:27017/productsdb'; // Example local DB
```

```
// Connect to MongoDB
mongoose.connect(MONGODB_URI)
    .then(() => console.log('Connected to MongoDB'))
    .catch(err => console.error('Could not connect to MongoDB:', err));

// Define a Mongoose Schema for a Product
const productSchema = new mongoose.Schema({
    name: {
        type: String,
        required: true,
        minlength: 3,
        maxlength: 255
    },
    price: {
        type: Number,
        required: true,
        min: 0
    },
    description: String,
    dateAdded: {
        type: Date,
        default: Date.now
    }
});

// Create a Mongoose Model from the schema
const Product = mongoose.model('Product', productSchema);

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

// --- REST API Endpoints (CRUD Operations) ---

// 1. CREATE a new product (POST /api/products)
app.post('/api/products', async (req, res) => {
    try {
        // Create a new Product instance from the request body
        const product = new Product({
            name: req.body.name,
            price: req.body.price,
            description: req.body.description
        });
        // Save the product to the database
        const result = await product.save();
        // Respond with the created product and 201 Created status
        res.status(201).json(result);
    } catch (error) {
        // Handle validation errors or other database errors
        res.status(400).json({ message: error.message });
    }
});

// 2. READ all products (GET /api/products)
app.get('/api/products', async (req, res) => {
    try {
        // Find all products in the database
        const products = await Product.find();
        // Respond with the array of products
        res.json(products);
    } catch (error) {
        res.status(500).json({ message: error.message });
    }
});

// 3. READ a single product by ID (GET /api/products/:id)
app.get('/api/products/:id', async (req, res) => {
```

```
    try {
        // Find a product by its ID
        const product = await Product.findById(req.params.id);
        if (!product) {
            return res.status(404).json({ message: 'Product not found.' });
        }
        res.json(product);
    } catch (error) {
        // Handle invalid ID format or other errors
        res.status(400).json({ message: 'Invalid product ID or ' + error.message
});
    }
});

// 4. UPDATE an existing product by ID (PUT /api/products/:id)
app.put('/api/products/:id', async (req, res) => {
    try {
        // Find the product by ID and update it with the new data
        // { new: true } returns the updated document
        const product = await Product.findByIdAndUpdate(
            req.params.id,
            {
                name: req.body.name,
                price: req.body.price,
                description: req.body.description
            },
            { new: true, runValidators: true } // runValidators ensures schema
validation on update
        );

        if (!product) {
            return res.status(404).json({ message: 'Product not found for
update.' });
        }
        res.json(product);
    } catch (error) {
        res.status(400).json({ message: error.message });
    }
});

// 5. DELETE a product by ID (DELETE /api/products/:id)
app.delete('/api/products/:id', async (req, res) => {
    try {
        // Find the product by ID and remove it
        const product = await Product.findByIdAndDelete(req.params.id);
        if (!product) {
            return res.status(404).json({ message: 'Product not found for
deletion.' });
        }
        // Respond with 204 No Content for successful deletion
        res.status(204).send();
    } catch (error) {
        res.status(500).json({ message: error.message });
    }
});

// --- Server Start ---

// Start the Express server
app.listen(PORT, () => {
    console.log(`MongoDB API server is running on http://localhost:${PORT}`);
    console.log('API Endpoints (use /api/products):');
    console.log('  POST   /api/products         - Create a new product');
    console.log('  GET    /api/products         - Get all products');
    console.log('  GET    /api/products/:id     - Get a product by ID');
    console.log('  PUT    /api/products/:id     - Update a product by ID');
    console.log('  DELETE /api/products/:id     - Delete a product by ID');
```

```
});
```

Input

You will use an API client (like Postman, Insomnia, or `curl`) to send HTTP requests.

1. **Create (POST):**
    - Method: `POST`
    - URL: `http://localhost:3000/api/products`
    - Headers: `Content-Type: application/json`
    - Body (raw JSON):
    - ```
      {
          "name": "Laptop Pro",
          "price": 1500,
          "description": "Powerful laptop for professionals."
      }
      ```
2. **Read All (GET):**
    - Method: `GET`
    - URL: `http://localhost:3000/api/products`
3. **Read One (GET by ID):**
    - Method: `GET`
    - URL: `http://localhost:3000/api/products/YOUR_PRODUCT_ID` (replace `YOUR_PRODUCT_ID` with an actual ID from a created product)
4. **Update (PUT):**
    - Method: `PUT`
    - URL: `http://localhost:3000/api/products/YOUR_PRODUCT_ID`
    - Headers: `Content-Type: application/json`
    - Body (raw JSON):
    - ```
      {
          "name": "Laptop Pro Max",
          "price": 1600,
          "description": "Updated and more powerful laptop."
      }
      ```
5. **Delete (DELETE):**
    - Method: `DELETE`
    - URL: `http://localhost:3000/api/products/YOUR_PRODUCT_ID`

Expected Output

1. **Create (POST /api/products):**
    - **Status:** `201 Created`
    - **Body:** (Example, `_id` will be a unique MongoDB ObjectId)
    - ```
      {
          "name": "Laptop Pro",
          "price": 1500,
          "description": "Powerful laptop for professionals.",
          "_id": "60c72b2f9b1d8c001c8e4d7a",
          "dateAdded": "2023-10-27T10:00:00.000Z",
          "__v": 0
      }
      ```
2. **Read All (GET /api/products):**
      ```

- **Status:** `200 OK`
- **Body:** (An array of product objects)

```
[
    {
        "_id": "60c72b2f9b1d8c001c8e4d7a",
        "name": "Laptop Pro",
        "price": 1500,
        "description": "Powerful laptop for professionals.",
        "dateAdded": "2023-10-27T10:00:00.000Z",
        "__v": 0
    }
    // ... other products
]
```

3. **Read One (GET /api/products/YOUR_PRODUCT_ID):**
   - **Status:** `200 OK`
   - **Body:** (A single product object)

```
{
    "_id": "60c72b2f9b1d8c001c8e4d7a",
    "name": "Laptop Pro",
    "price": 1500,
    "description": "Powerful laptop for professionals.",
    "dateAdded": "2023-10-27T10:00:00.000Z",
    "__v": 0
}
```

4. **Update (PUT /api/products/YOUR_PRODUCT_ID):**
   - **Status:** `200 OK`
   - **Body:** (The updated product object)

```
{
    "_id": "60c72b2f9b1d8c001c8e4d7a",
    "name": "Laptop Pro Max",
    "price": 1600,
    "description": "Updated and more powerful laptop.",
    "dateAdded": "2023-10-27T10:00:00.000Z",
    "__v": 0
}
```

5. **Delete (DELETE /api/products/YOUR_PRODUCT_ID):**
   - **Status:** `204 No Content`
   - **Body:** (Empty)

Lab 13: Create a Simple Login form using R

Title

Simple Login Form using R (Shiny App)

Aim

To create a basic web-based login form using R, specifically leveraging the Shiny framework to handle user input and provide a simple authentication mechanism.

Procedure

1. **Install R and RStudio:** If you don't have them, download and install R and RStudio.
2. **Install Shiny Package:** Open RStudio and install the Shiny package: `install.packages("shiny")`
3. **Create R Script:** Create a new R script file (e.g., `login_app.R`).
4. **Write Shiny App Code:** Add the R code for the Shiny application, defining the UI (login form) and server logic (authentication check), as shown in the "Source Code" section.
5. **Run the App:** In RStudio, open `login_app.R` and click the "Run App" button, or run `shiny::runApp('login_app.R')` in the console.
6. **Test Login:** Access the application in your web browser and try logging in with correct and incorrect credentials.

Source Code

login_app.R

```
# Load the Shiny library
library(shiny)

# Define UI for the login application
ui <- fluidPage(
    # Application title
    titlePanel("Simple Login Form"),

    # Sidebar layout for login elements
    sidebarLayout(
        sidebarPanel(
            # Input fields for username and password
            textInput("username", "Username:"),
            passwordInput("password", "Password:"),
            # Login button
            actionButton("login_button", "Login")
        ),
        # Main panel to display login status
        mainPanel(
            h3("Login Status:"),
            textOutput("login_status")
        )
    )
)

# Define server logic for the login application
server <- function(input, output) {

    # Reactive value to store login status message
    login_message <- reactiveVal("Please enter your credentials.")
```

```
    # Observe event when login button is clicked
    observeEvent(input$login_button, {
        # Define a simple hardcoded username and password for demonstration
        correct_username <- "user123"
        correct_password <- "password123"

        # Check if entered credentials match the correct ones
        if (input$username == correct_username && input$password ==
correct_password) {
            login_message("Login Successful! Welcome.")
        } else {
            login_message("Login Failed. Invalid username or password.")
        }
    })

    # Render the login status message
    output$login_status <- renderText({
        login_message()
    })
}

# Run the application
shinyApp(ui = ui, server = server)
```

Input

User enters a username and password into the text fields and clicks the "Login" button.

- **Correct Input:**
  - Username: `user123`
  - Password: `password123`
- **Incorrect Input:**
  - Any other combination of username/password.

Expected Output

A web application with a title "Simple Login Form", a "Username" input, a "Password" input, and a "Login" button. Below it, a "Login Status:" section.

- **Initial State:** "Login Status: Please enter your credentials."
- **After successful login (correct credentials):** "Login Status: Login Successful! Welcome."
- **After failed login (incorrect credentials):** "Login Status: Login Failed. Invalid username or password."

Lab 14: Making HTTP requests from Angular to an API

Title

Angular HTTP Requests to a REST API

Aim

To demonstrate how to make HTTP GET and POST requests from an Angular application to a backend REST API, using Angular's `HttpClient` module.

Procedure

1. **Set up Angular Project:**
    - Ensure Node.js and Angular CLI are installed.
    - Create a new Angular project: `ng new angular-api-app --no-standalone --skip-tests` (choose defaults for routing and stylesheet format).
    - Navigate into the project directory: `cd angular-api-app`.
2. **Set up Backend API (if not already running):**
    - You'll need a running backend API for this (e.g., the one from Lab 12). For demonstration, we'll assume a simple Express API running on `http://localhost:3000/api/data`.
    - **Create a simple `server.js` for testing (if you don't have Lab 12's API):**

```
const express = require('express');
const cors = require('cors'); // npm install cors
const app = express();
const PORT = 3000;

app.use(cors()); // Enable CORS for all origins
app.use(express.json());

let data = [{ id: 1, message: 'Hello from API!' }];
let nextId = 2;

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

app.post('/api/data', (req, res) => {
    const newItem = { id: nextId++, message: req.body.message };
    data.push(newItem);
    res.status(201).json(newItem);
});

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

    Run this `server.js` using `node server.js`.

3. **Configure Angular `HttpClient`:**
    - Open `src/app/app.module.ts` and import `HttpClientModule`. Add it to the `imports` array.
4. **Create a Service:**
    - Generate a service: `ng generate service data`.

- Implement `getData()` and `postData()` methods in `src/app/data.service.ts`.

5. **Update Component:**
   - Modify `src/app/app.component.ts` and `src/app/app.component.html` to use the service and display data/send requests.
6. **Run Angular App:**
   - Start the Angular development server: `ng serve --open`
7. **Test:** Interact with the Angular application in your browser.

Source Code

`server.js` (Simple Backend API for testing, if not using Lab 12's API)

```
const express = require('express');
const cors = require('cors'); // Required for cross-origin requests from Angular
const app = express();
const PORT = 3000;

// Enable CORS for all origins (important for development)
app.use(cors());
// Middleware to parse JSON request bodies
app.use(express.json());

// In-memory data store
let items = [
    { id: 1, text: 'Initial item from API' }
];
let nextItemId = 2;

// GET endpoint to retrieve all items
app.get('/api/items', (req, res) => {
    console.log('GET /api/items requested');
    res.json(items);
});

// POST endpoint to add a new item
app.post('/api/items', (req, res) => {
    const newItem = {
        id: nextItemId++,
        text: req.body.text || 'New item' // Get text from request body, default
if not provided
    };
    items.push(newItem);
    console.log('POST /api/items received:', newItem);
    res.status(201).json(newItem); // Respond with the created item and 201
status
});

// Start the server
app.listen(PORT, () => {
    console.log(`Backend API running on http://localhost:${PORT}`);
    console.log('Endpoints: GET /api/items, POST /api/items');
});
```

`src/app/app.module.ts`

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { HttpClientModule } from '@angular/common/http'; // Import
HttpClientModule
import { FormsModule } from '@angular/forms'; // For ngModel

import { AppComponent } from './app.component';

@NgModule({
  declarations: [
    AppComponent
```

```
  ],
  imports: [
    BrowserModule,
    HttpClientModule, // Add HttpClientModule here
    FormsModule // Add FormsModule for two-way data binding
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

src/app/data.service.ts

```
import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http'; // Import HttpClient
import { Observable } from 'rxjs'; // For reactive programming

@Injectable({
  providedIn: 'root'
})
export class DataService {
  // Define the base URL for your API
  private apiUrl = 'http://localhost:3000/api/items'; // Matches the server.js
endpoint

  constructor(private http: HttpClient) { } // Inject HttpClient

  // Method to make an HTTP GET request
  // Returns an Observable that emits the array of items
  getItems(): Observable<any[]> {
    return this.http.get<any[]>(this.apiUrl);
  }

  // Method to make an HTTP POST request
  // Takes an item object as input and sends it to the API
  // Returns an Observable that emits the created item
  addItem(item: { text: string }): Observable<any> {
    return this.http.post<any>(this.apiUrl, item);
  }
}
```

src/app/app.component.ts

```
import { Component, OnInit } from '@angular/core';
import { DataService } from './data.service'; // Import the DataService

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent implements OnInit {
  title = 'Angular API Client';
  items: any[] = []; // Array to store items fetched from the API
  newItemText: string = ''; // Model for the new item input field
  message: string = ''; // Message to display status/errors

  constructor(private dataService: DataService) { } // Inject the DataService

  ngOnInit(): void {
    this.fetchItems(); // Fetch items when the component initializes
  }

  // Method to fetch items from the API
  fetchItems(): void {
    this.message = 'Fetching items...';
    this.dataService.getItems().subscribe({
      next: (data) => {
```

```
        this.items = data;
        this.message = 'Items loaded successfully!';
      },
      error: (err) => {
        console.error('Error fetching items:', err);
        this.message = 'Failed to load items. Is the backend running?';
      }
    });
  }

  // Method to add a new item via API
  addItem(): void {
    if (this.newItemText.trim() === '') {
      this.message = 'Please enter text for the new item.';
      return;
    }

    this.message = 'Adding item...';
    this.dataService.addItem({ text: this.newItemText }).subscribe({
      next: (response) => {
        console.log('Item added:', response);
        this.newItemText = ''; // Clear input field
        this.fetchItems(); // Refresh the list after adding
        this.message = 'Item added successfully!';
      },
      error: (err) => {
        console.error('Error adding item:', err);
        this.message = 'Failed to add item.';
      }
    });
  }
}
```

src/app/app.component.html

```
<div class="container">
  <h1>{{ title }}</h1>

  <div class="input-section">
    <input type="text" [(ngModel)]="newItemText" placeholder="Enter new item
text">
    <button (click)="addItem()">Add Item</button>
  </div>

  <button (click)="fetchItems()" class="refresh-button">Refresh Items</button>

  <p class="status-message">{{ message }}</p>

  <h2>Items from API:</h2>
  <ul class="item-list">
    <li *ngIf="items.length === 0 && message !== 'Fetching items...'">No items
found.</li>
    <li *ngFor="let item of items">
      ID: {{ item.id }} - Text: {{ item.text }}
    </li>
  </ul>
</div>

<style>
  .container {
    font-family: 'Inter', sans-serif;
    max-width: 600px;
    margin: 40px auto;
    padding: 25px;
    border-radius: 12px;
    box-shadow: 0 6px 25px rgba(0, 0, 0, 0.15);
    background-color: #fff;
    text-align: center;
```

```
}

h1 {
  color: #007bff;
  margin-bottom: 25px;
}

.input-section {
  display: flex;
  gap: 10px;
  margin-bottom: 20px;
  justify-content: center;
}

input[type="text"] {
  flex-grow: 1;
  padding: 12px 15px;
  border: 1px solid #ced4da;
  border-radius: 8px;
  font-size: 1em;
  max-width: 300px; /* Limit input width */
}

button {
  background-color: #28a745;
  color: white;
  padding: 12px 20px;
  border: none;
  border-radius: 8px;
  cursor: pointer;
  font-size: 1em;
  transition: background-color 0.3s ease, transform 0.2s ease;
}

button:hover {
  background-color: #218838;
  transform: translateY(-2px);
}

.refresh-button {
  background-color: #007bff;
  margin-top: 15px;
  margin-bottom: 20px;
}

.refresh-button:hover {
  background-color: #0056b3;
}

.status-message {
  margin-top: 15px;
  font-style: italic;
  color: #555;
}

h2 {
  margin-top: 30px;
  color: #333;
  border-bottom: 1px solid #eee;
  padding-bottom: 10px;
}

.item-list {
  list-style: none;
  padding: 0;
  text-align: left;
}
```

```
  .item-list li {
    background-color: #f8f9fa;
    padding: 12px 15px;
    margin-bottom: 8px;
    border-radius: 8px;
    border: 1px solid #e9ecef;
    display: flex;
    align-items: center;
    justify-content: space-between;
  }
</style>
```

Input

1. **Initial Load:** The Angular application will automatically try to fetch items from the API when it loads.
2. **Add Item:** User types text into the input field and clicks "Add Item".
3. **Refresh:** User clicks the "Refresh Items" button.

Expected Output

1. **Initial Load:** The Angular application will display a heading "Angular API Client". Below it, a message like "Fetching items..." will appear, followed by "Items loaded successfully!" and a list of items fetched from the backend API (e.g., "ID: 1 - Text: Initial item from API").
2. **Add Item:** After typing text and clicking "Add Item", the new item will be sent to the API. Upon successful response, the input field will clear, the list of items will refresh, and the newly added item will appear in the list (e.g., "ID: 2 - Text: My new item"). A status message like "Item added successfully!" will be displayed.
3. **Refresh:** Clicking "Refresh Items" will re-fetch the current list of items from the API, updating the display.

Lab 15: More complex views and routing parameters

Title

Angular Complex Views and Routing Parameters

Aim

To create an Angular application with multiple views (components) and implement client-side routing, including the use of routing parameters to pass data between views.

Procedure

1. **Set up Angular Project:**
   - Ensure Node.js and Angular CLI are installed.
   - Create a new Angular project: `ng new angular-routing-app --no-standalone --skip-tests` (choose `Yes` for Angular routing, and `CSS` for stylesheet format).
   - Navigate into the project directory: `cd angular-routing-app`.
2. **Generate Components:**
   - Generate three components: `Home`, `Products`, and `ProductDetail`.
     - `ng generate component home`
     - `ng generate component products`
     - `ng generate component product-detail`
3. **Configure Routing:**
   - Open `src/app/app-routing.module.ts`.
   - Define routes for `Home`, `Products`, and `ProductDetail` (with a parameter).
4. **Implement Component Logic:**
   - In `HomeComponent`, add simple welcome text.
   - In `ProductsComponent`, list some static products and provide links to `ProductDetailComponent`.
   - In `ProductDetailComponent`, use `ActivatedRoute` to retrieve the routing parameter (product ID) and display product details.
5. **Update `AppComponent`:**
   - Modify `src/app/app.component.html` to include navigation links and the `router-outlet`.
6. **Run Angular App:**
   - Start the Angular development server: `ng serve --open`
7. **Test Routing:** Navigate between different views using the links and by directly entering URLs with parameters.

Source Code

src/app/app-routing.module.ts

```
import { NgModule } from '@angular/core';
import { RouterModule, Routes } from '@angular/router';
import { HomeComponent } from './home/home.component';
import { ProductsComponent } from './products/products.component';
import { ProductDetailComponent } from './product-detail/product-
detail.component';

// Define the application routes
const routes: Routes = [
```

```
  { path: '', redirectTo: '/home', pathMatch: 'full' }, // Default route
redirects to /home
  { path: 'home', component: HomeComponent }, // Route for the Home page
  { path: 'products', component: ProductsComponent }, // Route for the Products
list page
  // Route for product detail, with a dynamic 'id' parameter
  { path: 'products/:id', component: ProductDetailComponent },
  { path: '**', redirectTo: '/home' } // Wildcard route for any unmatched URL,
redirects to home
];

@NgModule({
  imports: [RouterModule.forRoot(routes)], // Configure the router at the
application root
  exports: [RouterModule] // Export RouterModule to make router directives
available to other modules
})
export class AppRoutingModule { }
```

src/app/app.component.html

```
<nav class="navbar">
  <a routerLink="/home" routerLinkActive="active">Home</a>
  <a routerLink="/products" routerLinkActive="active">Products</a>
</nav>

<div class="content">
  <router-outlet></router-outlet>
</div>

<style>
  body {
    font-family: 'Inter', sans-serif;
    margin: 0;
    background-color: #f4f4f4;
    color: #333;
  }

  .navbar {
    background-color: #333;
    padding: 15px 20px;
    text-align: center;
    box-shadow: 0 2px 5px rgba(0,0,0,0.2);
  }

  .navbar a {
    color: white;
    text-decoration: none;
    padding: 10px 20px;
    margin: 0 10px;
    border-radius: 8px;
    transition: background-color 0.3s ease;
  }

  .navbar a:hover {
    background-color: #555;
  }

  .navbar a.active {
    background-color: #007bff;
    font-weight: bold;
  }

  .content {
    padding: 20px;
    max-width: 800px;
    margin: 20px auto;
    background-color: #fff;
```

```
    border-radius: 12px;
    box-shadow: 0 4px 15px rgba(0,0,0,0.1);
  }

  h2 {
    color: #007bff;
    margin-bottom: 20px;
    text-align: center;
  }

  ul {
    list-style: none;
    padding: 0;
  }

  li {
    background-color: #e9ecef;
    padding: 15px;
    margin-bottom: 10px;
    border-radius: 8px;
    display: flex;
    justify-content: space-between;
    align-items: center;
    transition: background-color 0.3s ease;
  }

  li:hover {
    background-color: #d1e7dd;
  }

  li a {
    text-decoration: none;
    color: #333;
    font-weight: bold;
  }

  li a:hover {
    color: #007bff;
  }

  .back-button {
    display: inline-block;
    background-color: #6c757d;
    color: white;
    padding: 10px 15px;
    border-radius: 8px;
    text-decoration: none;
    margin-top: 20px;
    transition: background-color 0.3s ease;
  }

  .back-button:hover {
    background-color: #5a6268;
  }
</style>
```

src/app/home/home.component.ts

```
import { Component } from '@angular/core';

@Component({
  selector: 'app-home',
  template: `
    <h2>Welcome to the Home Page!</h2>
    <p>This is a simple Angular application demonstrating routing and complex
views.</p>
    <p>Use the navigation above to explore different sections.</p>
  `,
```

```
  styles: [`
    h2 { text-align: center; color: #28a745; }
    p { text-align: center; margin-bottom: 10px; }
  `]
})
export class HomeComponent { }
```

src/app/products/products.component.ts

```
import { Component, OnInit } from '@angular/core';

@Component({
  selector: 'app-products',
  template: `
    <h2>Our Products</h2>
    <ul>
      <li *ngFor="let product of products">
        <span>{{ product.name }}</span>
        <a [routerLink]="['/products', product.id]">View Details</a>
      </li>
    </ul>
  `,
  styles: [`
    h2 { text-align: center; color: #007bff; }
    ul { list-style: none; padding: 0; }
    li {
      background-color: #f8f9fa;
      padding: 15px;
      margin-bottom: 10px;
      border-radius: 8px;
      display: flex;
      justify-content: space-between;
      align-items: center;
      box-shadow: 0 2px 5px rgba(0,0,0,0.05);
    }
    li a {
      background-color: #28a745;
      color: white;
      padding: 8px 12px;
      border-radius: 5px;
      text-decoration: none;
      transition: background-color 0.3s ease;
    }
    li a:hover {
      background-color: #218838;
    }
  `]
})
export class ProductsComponent implements OnInit {
  products = [
    { id: 1, name: 'Laptop Pro', description: 'High-performance laptop.' },
    { id: 2, name: 'Wireless Mouse', description: 'Ergonomic and precise.' },
    { id: 3, name: 'Mechanical Keyboard', description: 'Tactile typing
experience.' },
    { id: 4, name: 'External SSD', description: 'Fast and portable storage.' }
  ];

  constructor() { }

  ngOnInit(): void {
  }
}
```

src/app/product-detail/product-detail.component.ts

```
import { Component, OnInit } from '@angular/core';
import { ActivatedRoute, Router } from '@angular/router'; // Import
ActivatedRoute and Router
```

```
@Component({
  selector: 'app-product-detail',
  template: `
    <h2>Product Details</h2>
    <div *ngIf="product" class="product-card">
      <h3>{{ product.name }}</h3>
      <p><strong>ID:</strong> {{ product.id }}</p>
      <p><strong>Description:</strong> {{ product.description }}</p>
    </div>
    <div *ngIf="!product" class="no-product">
      <p>Product not found.</p>
    </div>
    <button (click)="goBack()" class="back-button">Back to Products</button>
  `,
  styles: [`
    h2 { text-align: center; color: #007bff; margin-bottom: 25px; }
    .product-card {
      background-color: #f0f8ff;
      padding: 20px;
      border-radius: 10px;
      box-shadow: 0 2px 10px rgba(0,0,0,0.1);
      margin-bottom: 20px;
    }
    .product-card h3 {
      color: #333;
      margin-top: 0;
      margin-bottom: 15px;
      font-size: 1.5em;
    }
    .product-card p {
      margin-bottom: 8px;
    }
    .no-product {
      text-align: center;
      color: #dc3545;
      font-weight: bold;
    }
    .back-button {
      display: block;
      margin: 0 auto;
      background-color: #6c757d;
      color: white;
      padding: 10px 15px;
      border-radius: 8px;
      text-decoration: none;
      border: none;
      cursor: pointer;
      transition: background-color 0.3s ease;
    }
    .back-button:hover {
      background-color: #5a6268;
    }
  `]
})
export class ProductDetailComponent implements OnInit {
  product: any; // To store the fetched product details

  // Dummy product data (replace with API call in a real app)
  private products = [
    { id: 1, name: 'Laptop Pro', description: 'High-performance laptop with 16GB
RAM and 512GB SSD.' },
    { id: 2, name: 'Wireless Mouse', description: 'Ergonomic mouse with
adjustable DPI and long battery life.' },
    { id: 3, name: 'Mechanical Keyboard', description: 'RGB backlit keyboard
with clicky blue switches.' },
```

```
    { id: 4, name: 'External SSD', description: '1TB portable SSD with USB-C 3.2
Gen 2 speed.' }
  ];

  constructor(
    private route: ActivatedRoute, // To access route parameters
    private router: Router // To navigate programmatically
  ) { }

  ngOnInit(): void {
    // Subscribe to route parameter changes
    this.route.paramMap.subscribe(params => {
      const productId = +params.get('id')!; // Get 'id' parameter and convert to
number
      this.product = this.products.find(p => p.id === productId); // Find
product by ID
    });
  }

  goBack(): void {
    this.router.navigate(['/products']); // Navigate back to the products list
  }
}
```

Input

User navigates the application by:

1. Clicking "Home" in the navigation bar.
2. Clicking "Products" in the navigation bar.
3. Clicking "View Details" next to a product on the Products page.
4. Directly entering URLs like `http://localhost:4200/products/1` or `http://localhost:4200/products/3`.
5. Clicking "Back to Products" on the Product Details page.

Expected Output

1. **Home Page (`/home`):** Displays "Welcome to the Home Page!" and descriptive text.
2. **Products Page (`/products`):** Displays "Our Products" heading and a list of product names, each with a "View Details" link.
3. **Product Detail Page (`/products/:id`):**
    - If a valid product ID is provided (e.g., `/products/1`), it displays "Product Details" heading, and the name, ID, and description of that specific product.
    - If an invalid product ID is provided (e.g., `/products/99`), it displays "Product not found."
    - A "Back to Products" button will navigate back to the products list.