SRM Institute of Science and Technology

Department of Computer Applications

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

Circular – 2020-21

MCA 1st semester
ADVANCED WEB APPLICATION DEVELOPMENT (PCA20D01J)

Lab Manual

Lab 1: Sample Application

Title: Building a Basic Sample Web Application

Aim: To understand the fundamental structure of a web application and set up a simple "Hello World" style application.

Procedure:

1. Initialize a new project directory.
2. Create a basic HTML file (`index.html`).
3. Add simple content, e.g., a heading with "Hello, World!".
4. Open the HTML file in a web browser to verify.
5. (Optional) Set up a basic local server (e.g., using Node.js `http-server` or Python `SimpleHTTPServer`) to serve the file.

Source Code:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Sample Application</title>
</head>
<body>
    <h1>Hello, Advanced Web Application Development!</h1>
    <p>This is a basic sample application.</p>
</body>
</html>
```

Input: N/A (Directly opening the HTML file or accessing via a local server URL).

Expected Output: A web page displaying "Hello, Advanced Web Application Development!" and "This is a basic sample application." in the browser.

Lab 2: Planning a Real Application

Title: Conceptualizing and Planning a Real-World Web Application

Aim: To develop a high-level understanding of the application development lifecycle, including requirements gathering, feature identification, and basic architectural design.

Procedure:

1. Choose a simple real-world application idea (e.g., a simple To-Do list, a blog, a small e-commerce product catalog).
2. Define the core functionalities (e.g., for a To-Do list: add task, mark as complete, delete task).
3. Identify key user roles and their interactions.
4. Sketch out a basic user interface (UI) flow.
5. List the potential technologies/frameworks to be used (e.g., Node.js, Express, Angular, MongoDB).

Source Code: N/A (This lab is primarily conceptual and involves documentation, diagrams, or wireframes rather than code).

Input: Project requirements, user stories, use cases.

Expected Output: A documented plan including application features, user flow diagrams, and a technology stack proposal.

Lab 3: Development Hardware

Title: Understanding and Setting Up Development Environment Hardware/Software

Aim: To identify the necessary hardware and software components for web application development and configure a suitable development environment.

Procedure:

1. List essential hardware components (e.g., computer with sufficient RAM, storage, monitor).
2. Identify and install core software tools (e.g., Operating System, Code Editor/IDE like VS Code, Web Browser, Node.js, Git, Database system like MongoDB or PostgreSQL).
3. Configure environment variables and paths as required for installed tools.
4. Verify installations by running simple commands (e.g., `node -v`, `npm -v`, `git --version`).

Source Code: N/A (This lab focuses on environment setup. Any "code" would be configuration files or shell scripts).

Input: System requirements, installation guides.

Expected Output: A fully configured development environment with all necessary software installed and accessible.

Lab 4: How to Move Data from View to the Controller

Title: Implementing Data Flow from Frontend View to Backend Controller

Aim: To understand and implement mechanisms for sending data from a client-side (view) to a server-side (controller) using HTTP requests.

Procedure:

1. Set up a basic server-side application (e.g., using Express.js).
2. Create an HTML form on the client-side with input fields.
3. Use JavaScript (e.g., `fetch` API or `XMLHttpRequest`) to capture form data on submission.
4. Send the captured data to a specific API endpoint on the server using POST or GET requests.
5. On the server, define a route handler (controller) to receive and process the incoming data.

Source Code:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Data Transfer</title>
</head>
<body>
    <h1>Submit Data</h1>
    <form id="myForm">
        <label for="name">Name:</label>
        <input type="text" id="name" name="name" required><br><br>
        <label for="email">Email:</label>
        <input type="email" id="email" name="email" required><br><br>
        <button type="submit">Submit</button>
    </form>

    <script>
        document.getElementById('myForm').addEventListener('submit', async
function(event) {
            event.preventDefault();
            const name = document.getElementById('name').value;
            const email = document.getElementById('email').value;

            try {
                const response = await fetch('/submit-data', {
                    method: 'POST',
                    headers: {
                        'Content-Type': 'application/json'
                    },
                    body: JSON.stringify({ name, email })
                });
                const result = await response.json();
                console.log('Server Response:', result);
                alert('Data submitted successfully! Check console for
response.');
            } catch (error) {
                console.error('Error submitting data:', error);
                alert('Failed to submit data.');
```

```
            }
        });
    </script>
</body>
</html>
```javascript
// app.js (Backend Controller - Express.js)
const express = require('express');
const bodyParser = require('body-parser');
const app = express();
const port = 3000;

// Middleware to parse JSON bodies
app.use(bodyParser.json());
// Serve static files (like index.html)
app.use(express.static(__dirname));

app.post('/submit-data', (req, res) => {
    const { name, email } = req.body;
    console.log('Received data:', { name, email });
    // In a real application, you would save this data to a database
    res.json({ message: 'Data received successfully!', data: { name, email }
});
});

app.listen(port, () => {
    console.log(`Server listening at http://localhost:${port}`);
});
```

Input: Text entered into the "Name" and "Email" fields on the web form.

Expected Output:

Browser: An alert confirming successful submission.

Server Console: Log message showing the received name and email.

Browser Console: JSON response from the server.
```

Lab 5: Setting up the HTML Framework with Jade templates and Bootstrap

Title: Structuring Web Pages with Pug (Jade) Templates and Styling with Bootstrap

Aim: To learn how to use a templating engine (Pug, formerly Jade) for dynamic HTML generation and apply Bootstrap for responsive and aesthetically pleasing UI design.

Procedure:

1. Install Pug and Bootstrap in your project.
2. Configure your server-side framework (e.g., Express) to use Pug as the view engine.
3. Create a basic Pug template file (`layout.pug`) for common page structure.
4. Create a specific Pug file (`index.pug`) that extends the layout and includes Bootstrap CSS/JS.
5. Add some basic Bootstrap components (e.g., navigation bar, cards) to the Pug template.
6. Render the Pug template from a server route.

Source Code:

```
// layout.pug (Pug Layout Template)
doctype html
html(lang="en")
  head
    meta(charset="UTF-8")
    meta(name="viewport", content="width=device-width, initial-scale=1.0")
    title #{title}
    // Bootstrap CSS CDN

link(href="[https://cdn.jsdelivr.net/npm/bootstrap@5.3.0/dist/css/bootstrap.m
in.css](https://cdn.jsdelivr.net/npm/bootstrap@5.3.0/dist/css/bootstrap.min.c
ss)", rel="stylesheet")
  body
    block content
    // Bootstrap JS CDN (Bundle with Popper)

script(src="[https://cdn.jsdelivr.net/npm/bootstrap@5.3.0/dist/js/bootstrap.b
undle.min.js](https://cdn.jsdelivr.net/npm/bootstrap@5.3.0/dist/js/bootstrap.
bundle.min.js)")
```pug
// index.pug (Pug Page Template)
extends layout.pug

block content
  nav.navbar.navbar-expand-lg.navbar-dark.bg-primary
    .container-fluid
      a.navbar-brand(href="#") My App
      button.navbar-toggler(type="button", data-bs-toggle="collapse", data-
bs-target="#navbarNav", aria-controls="navbarNav", aria-expanded="false",
aria-label="Toggle navigation")
        span.navbar-toggler-icon
      #navbarNav.collapse.navbar-collapse
        ul.navbar-nav
          li.nav-item
            a.nav-link.active(aria-current="page", href="#") Home
          li.nav-item
            a.nav-link(href="#") Features
```
```

```
  .container.mt-4
    h1.mb-3 Welcome to the Pug & Bootstrap App!
    .card
      .card-header Featured
      .card-body
        h5.card-title This is a Bootstrap Card
        p.card-text With some quick example text to build on the card title
and make up the bulk of the card's content.
        a.btn.btn-primary(href="#") Go somewhere
```

```javascript
// app.js (Express.js Server)
const express = require('express');
const path = require('path');
const app = express();
const port = 3000;

// Configure Pug as the view engine
app.set('views', path.join(__dirname, 'views'));
app.set('view engine', 'pug');

app.get('/', (req, res) => {
    res.render('index', { title: 'Pug & Bootstrap Demo' });
});

app.listen(port, () => {
    console.log(`Server listening at http://localhost:${port}`);
});
```

Input: N/A (Accessing the root URL of the server).

Expected Output: A web page rendered with a Bootstrap-styled navigation bar, a heading, and a Bootstrap card, all generated using Pug templates.

Lab 6: Take the Data out of the Views and Make them Smarter

Title: Separating Data from Views and Implementing Dynamic Content Rendering

Aim: To improve application architecture by fetching data from a separate source (e.g., a mock API or a local data structure) and dynamically injecting it into the view using templating engine features.

Procedure:

1. Create a JavaScript file or a simple array to hold mock data (e.g., a list of products or users).
2. Modify the server-side route to retrieve this data.
3. Pass the retrieved data as an object to the Pug `res.render()` method.
4. Update the Pug template to iterate over the passed data and display it dynamically (e.g., using Pug's `each` loop).

Source Code:

```
// data.js (Mock Data)
const products = [
    { id: 1, name: 'Laptop', price: 1200, description: 'Powerful computing on
the go.' },
    { id: 2, name: 'Mouse', price: 25, description: 'Ergonomic wireless
mouse.' },
    { id: 3, name: 'Keyboard', price: 75, description: 'Mechanical keyboard
with RGB.' }
];
module.exports = products;
```pug
// products.pug (Pug Template to display products)
extends layout.pug

block content
  .container.mt-4
    h1.mb-3 Our Products
    .row
      each product in products
        .col-md-4.mb-4
          .card
            .card-body
              h5.card-title #{product.name}
              h6.card-subtitle.mb-2.text-muted $#{product.price}
              p.card-text #{product.description}
              a.btn.btn-info.btn-sm(href="#") View Details
```javascript
// app.js (Express.js Server)
const express = require('express');
const path = require('path');
const products = require('./data'); // Import mock data
const app = express();
const port = 3000;

app.set('views', path.join(__dirname, 'views'));
app.set('view engine', 'pug');

app.get('/products', (req, res) => {
```

```
    res.render('products', { title: 'Product List', products: products });
});

app.listen(port, () => {
    console.log(`Server listening at http://localhost:${port}`);
});
```

Input: N/A (Accessing the `/products` URL).

Expected Output: A web page displaying a list of products, where each product's name, price, and description are dynamically rendered from the `products` data array.

Lab 7: Pushing up the Data

Title: Implementing Data Persistence: Saving Data to a Database

Aim: To learn how to connect a web application to a database and persist data submitted from the frontend.

Procedure:

1. Choose and set up a database (e.g., MongoDB, PostgreSQL, SQLite).
2. Install the necessary database driver/ORM for your backend language (e.g., Mongoose for MongoDB, Sequelize for SQL databases).
3. Establish a connection to the database from your server-side application.
4. Modify the controller (from Lab 4) to save the incoming data to the database instead of just logging it.
5. Implement a mechanism to retrieve and display the saved data.

Source Code: (Example using MongoDB with Mongoose)

```
// app.js (Express.js Server with MongoDB/Mongoose)
const express = require('express');
const bodyParser = require('body-parser');
const mongoose = require('mongoose');
const app = express();
const port = 3000;

// Connect to MongoDB
mongoose.connect('mongodb://localhost:27017/myappdb', {
    useNewUrlParser: true,
    useUnifiedTopology: true
})
.then(() => console.log('MongoDB connected...'))
.catch(err => console.error(err));

// Define a simple Schema and Model
const itemSchema = new mongoose.Schema({
    name: String,
    description: String,
    date: { type: Date, default: Date.now }
});
const Item = mongoose.model('Item', itemSchema);

app.use(bodyParser.json());
app.use(express.static(__dirname)); // Serve static files

// Route to save data
app.post('/save-item', async (req, res) => {
    const { name, description } = req.body;
    try {
        const newItem = new Item({ name, description });
        await newItem.save();
        res.status(201).json({ message: 'Item saved successfully!', item:
newItem });
    } catch (error) {
        console.error('Error saving item:', error);
        res.status(500).json({ message: 'Failed to save item.' });
    }
});
```

```
// Route to get all items
app.get('/items', async (req, res) => {
    try {
        const items = await Item.find();
        res.json(items);
    } catch (error) {
        console.error('Error fetching items:', error);
        res.status(500).json({ message: 'Failed to fetch items.' });
    }
});

app.listen(port, () => {
    console.log(`Server listening at http://localhost:${port}`);
});
```
```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Save Data</title>
</head>
<body>
    <h1>Add New Item</h1>
    <form id="itemForm">
        <label for="itemName">Item Name:</label>
        <input type="text" id="itemName" required><br><br>
        <label for="itemDesc">Description:</label>
        <textarea id="itemDesc"></textarea><br><br>
        <button type="submit">Save Item</button>
    </form>

    <h2>Saved Items</h2>
    <ul id="itemList"></ul>

    <script>
        async function fetchItems() {
            try {
                const response = await fetch('/items');
                const items = await response.json();
                const itemList = document.getElementById('itemList');
                itemList.innerHTML = ''; // Clear previous list
                items.forEach(item => {
                    const li = document.createElement('li');
                    li.textContent = `${item.name}: ${item.description}
(Added: ${new Date(item.date).toLocaleDateString()})`;
                    itemList.appendChild(li);
                });
            } catch (error) {
                console.error('Error fetching items:', error);
            }
        }

        document.getElementById('itemForm').addEventListener('submit', async
function(event) {
            event.preventDefault();
            const name = document.getElementById('itemName').value;
            const description = document.getElementById('itemDesc').value;

            try {
                const response = await fetch('/save-item', {
                    method: 'POST',
                    headers: { 'Content-Type': 'application/json' },
                    body: JSON.stringify({ name, description })
                });
                const result = await response.json();
                console.log('Save Response:', result);
```

```
                alert(result.message);
                document.getElementById('itemForm').reset(); // Clear form
                fetchItems(); // Refresh the list
            } catch (error) {
                console.error('Error saving item:', error);
                alert('Failed to save item.');
            }
        });

        // Fetch items on page load
        window.onload = fetchItems;
    </script>
</body>
</html>
```

Input: Text entered into the "Item Name" and "Description" fields.

Expected Output:

Successful saving of data to the connected database.

Confirmation message in the browser.

The list of saved items on the web page updates dynamically to include the newly added item.

Lab 8: Making the Application Use the Right Database

Title: Selecting and Integrating an Appropriate Database System

Aim: To understand different types of databases (SQL vs. NoSQL) and choose the most suitable one for a given application, then integrate it effectively.

Procedure:

1. Review the requirements of the application (e.g., structured data, flexible schema, scalability needs).
2. Compare relational databases (e.g., MySQL, PostgreSQL) with NoSQL databases (e.g., MongoDB, Cassandra, Redis).
3. Justify the choice of database for the planned application.
4. If different from previous labs, set up and configure the chosen database system.
5. Migrate or adapt the data persistence logic (from Lab 7) to work with the new database.

Source Code: (This lab involves adapting the code from Lab 7 based on the chosen database. For example, if switching from MongoDB to PostgreSQL, the Mongoose code would be replaced with `pg` or Sequelize code.)

Example (Conceptual change):

```
// Example: Switching to PostgreSQL with 'pg'
const { Pool } = require('pg');
const pool = new Pool({
    user: 'your_user',
    host: 'localhost',
    database: 'your_db',
    password: 'your_password',
    port: 5432,
});

// ... in your route handler ...
app.post('/save-item', async (req, res) => {
    const { name, description } = req.body;
    try {
        const result = await pool.query(
            'INSERT INTO items (name, description) VALUES ($1, $2)
RETURNING *',
            [name, description]
        );
        res.status(201).json({ message: 'Item saved successfully!',
item: result.rows[0] });
    } catch (error) {
        console.error('Error saving item:', error);
        res.status(500).json({ message: 'Failed to save item.' });
    }
});
```

Input: Application requirements, database system choices.

Expected Output: A functional web application connected to and utilizing the chosen database system, demonstrating appropriate data storage and retrieval.

Lab 9: Setting up the API in Express

Title: Designing and Implementing RESTful APIs with Express.js

Aim: To create a robust backend API using Express.js that exposes endpoints for various data operations (CRUD: Create, Read, Update, Delete).

Procedure:

1. Set up an Express.js application.
2. Define API routes for resources (e.g., `/api/products`, `/api/users`).
3. Implement HTTP methods (GET, POST, PUT/PATCH, DELETE) for each resource.
4. Connect these routes to database operations (from Lab 7/8) to handle data.
5. Use middleware (e.g., `body-parser`) for parsing request bodies.
6. Test API endpoints using tools like Postman or Insomnia.

Source Code: (Building upon Lab 7's `app.js` with more comprehensive API routes)

```
// app.js (Express.js REST API)
const express = require('express');
const bodyParser = require('body-parser');
const mongoose = require('mongoose'); // Or your chosen DB library
const app = express();
const port = 3000;

// Connect to DB (as in Lab 7/8)
mongoose.connect('mongodb://localhost:27017/myappdb')
    .then(() => console.log('MongoDB connected for API...'))
    .catch(err => console.error(err));

const itemSchema = new mongoose.Schema({
    name: String,
    description: String,
    date: { type: Date, default: Date.now }
});
const Item = mongoose.model('Item', itemSchema);

app.use(bodyParser.json());

// API Routes for /api/items
// GET all items
app.get('/api/items', async (req, res) => {
    try {
        const items = await Item.find();
        res.json(items);
    } catch (error) {
        res.status(500).json({ message: error.message });
    }
});

// GET a single item by ID
app.get('/api/items/:id', async (req, res) => {
    try {
        const item = await Item.findById(req.params.id);
        if (!item) return res.status(404).json({ message: 'Item not found'
});
        res.json(item);
    } catch (error) {
        res.status(500).json({ message: error.message });
```

```
    }
});

// POST a new item
app.post('/api/items', async (req, res) => {
    const newItem = new Item({
        name: req.body.name,
        description: req.body.description
    });
    try {
        const savedItem = await newItem.save();
        res.status(201).json(savedItem);
    } catch (error) {
        res.status(400).json({ message: error.message });
    }
});

// PUT/PATCH update an item
app.patch('/api/items/:id', async (req, res) => {
    try {
        const updatedItem = await Item.findByIdAndUpdate(
            req.params.id,
            req.body,
            { new: true } // Return the updated document
        );
        if (!updatedItem) return res.status(404).json({ message: 'Item not
found' });
        res.json(updatedItem);
    } catch (error) {
        res.status(400).json({ message: error.message });
    }
});

// DELETE an item
app.delete('/api/items/:id', async (req, res) => {
    try {
        const deletedItem = await Item.findByIdAndDelete(req.params.id);
        if (!deletedItem) return res.status(404).json({ message: 'Item not
found' });
        res.json({ message: 'Item deleted successfully' });
    } catch (error) {
        res.status(500).json({ message: error.message });
    }
});

app.listen(port, () => {
    console.log(`API Server listening at http://localhost:${port}`);
});
```

Input: HTTP requests (GET, POST, PUT, DELETE) to API endpoints, potentially with JSON request bodies for POST/PUT.

Expected Output:

GET: JSON array of items or a single item.

POST: JSON object of the newly created item with status 201.

PUT/PATCH: JSON object of the updated item.

DELETE: Success message JSON object.

Appropriate HTTP status codes (200, 201, 400, 404, 500).

Lab 10: Building the API Request

Title: Consuming RESTful APIs from a Frontend Application

Aim: To learn how to make HTTP requests from a client-side application (e.g., using plain JavaScript, jQuery, or a frontend framework) to interact with a backend API.

Procedure:

1. Create a simple HTML page with buttons or forms for interacting with the API (e.g., "Fetch Items", "Add Item").
2. Use JavaScript's `fetch` API (or `axios` library) to send requests to the API endpoints defined in Lab 9.
3. Handle different HTTP methods (GET, POST, etc.) and include request bodies/headers as needed.
4. Process the JSON responses from the API and update the frontend UI accordingly.
5. Implement basic error handling for API calls.

Source Code: (Frontend part to interact with the API from Lab 9)

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>API Client</title>
    <style>
        body { font-family: sans-serif; margin: 20px; }
        #output { border: 1px solid #ccc; padding: 10px; min-height: 100px;
margin-top: 20px; }
        button { margin-right: 10px; padding: 8px 15px; cursor: pointer; }
        input, textarea { width: 300px; padding: 5px; margin-bottom: 10px; }
    </style>
</head>
<body>
    <h1>API Interaction</h1>

    <button id="fetchItemsBtn">Fetch All Items</button>
    <button id="fetchSingleItemBtn">Fetch Item by ID</button>
    <input type="text" id="itemId" placeholder="Item ID for single fetch">

    <h3>Add New Item</h3>
    <input type="text" id="newItemName" placeholder="Item Name">
    <textarea id="newItemDesc" placeholder="Item Description"></textarea>
    <button id="addItemBtn">Add Item</button>

    <h3>Update Item</h3>
    <input type="text" id="updateItemId" placeholder="Item ID to update">
    <input type="text" id="updateItemName" placeholder="New Name (optional)">
    <textarea id="updateItemDesc" placeholder="New Description
(optional)"></textarea>
    <button id="updateItemBtn">Update Item</button>

    <h3>Delete Item</h3>
    <input type="text" id="deleteItemId" placeholder="Item ID to delete">
    <button id="deleteItemBtn">Delete Item</button>

    <h2>API Output:</h2>
    <pre id="output"></pre>
```

```
    <script>
        const outputDiv = document.getElementById('output');

        async function makeApiRequest(url, method = 'GET', data = null) {
            const options = {
                method: method,
                headers: {
                    'Content-Type': 'application/json',
                },
            };
            if (data) {
                options.body = JSON.stringify(data);
            }

            try {
                const response = await fetch(url, options);
                const result = await response.json();
                outputDiv.textContent = JSON.stringify(result, null, 2);
                if (!response.ok) {
                    alert(`API Error: ${result.message ||
response.statusText}`);
                }
                return result;
            } catch (error) {
                outputDiv.textContent = `Error: ${error.message}`;
                alert(`Network Error: ${error.message}`);
            }
        }

        document.getElementById('fetchItemsBtn').addEventListener('click', ()
=> {
            makeApiRequest('/api/items');
        });


document.getElementById('fetchSingleItemBtn').addEventListener('click', () =>
{
            const id = document.getElementById('itemId').value;
            if (id) {
                makeApiRequest(`/api/items/${id}`);
            } else {
                alert('Please enter an Item ID.');
            }
        });

        document.getElementById('addItemBtn').addEventListener('click', () =>
{
            const name = document.getElementById('newItemName').value;
            const description = document.getElementById('newItemDesc').value;
            if (name && description) {
                makeApiRequest('/api/items', 'POST', { name, description });
            } else {
                alert('Please enter both name and description for the new
item.');
            }
        });

        document.getElementById('updateItemBtn').addEventListener('click', ()
=> {
            const id = document.getElementById('updateItemId').value;
            const name = document.getElementById('updateItemName').value;
            const description =
document.getElementById('updateItemDesc').value;
            if (id) {
                const updateData = {};
                if (name) updateData.name = name;
```

```
                if (description) updateData.description = description;
                if (Object.keys(updateData).length > 0) {
                    makeApiRequest(`/api/items/${id}`, 'PATCH', updateData);
                } else {
                    alert('Please enter at least one field (name or
description) to update.');
                }
            } else {
                alert('Please enter the Item ID to update.');
            }
        });

        document.getElementById('deleteItemBtn').addEventListener('click', ()
=> {
            const id = document.getElementById('deleteItemId').value;
            if (id) {
                if (confirm('Are you sure you want to delete this item?')) {
                    makeApiRequest(`/api/items/${id}`, 'DELETE');
                }
            } else {
                alert('Please enter the Item ID to delete.');
            }
        });
    </script>
</body>
</html>
```

Input: User clicks on buttons, enters text into input fields for item creation/update/deletion, and provides item IDs.

Expected Output:

The `output` div on the page displays the JSON response from the API.

Actions (adding, updating, deleting) are reflected in the backend database.

Error messages are displayed via `alert()` if API calls fail.

Lab 11: Displaying and Filtering the Homepage List

Title: Rendering Dynamic Lists and Implementing Client-Side Filtering

Aim: To display a list of items fetched from an API on the homepage and enable users to filter this list based on various criteria (e.g., search term, category).

Procedure:

1. On the homepage (e.g., `index.html`), fetch a list of items from your API (e.g., `/api/items`).
2. Dynamically render this list using JavaScript (e.g., creating `div` or `li` elements for each item).
3. Add an input field for a search query and/or dropdowns for filtering options.
4. Implement JavaScript logic to filter the displayed list in real-time as the user types or selects filter options, without making new API calls for every filter change (client-side filtering).

Source Code: (HTML and JavaScript for displaying and filtering a list)

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Homepage List</title>
    <style>
        body { font-family: sans-serif; margin: 20px; }
        #filterInput { margin-bottom: 20px; padding: 8px; width: 300px; }
        .item-card { border: 1px solid #eee; padding: 10px; margin-bottom:
10px; border-radius: 5px; }
        .item-card h3 { margin-top: 0; }
    </style>
</head>
<body>
    <h1>Our Items</h1>
    <input type="text" id="filterInput" placeholder="Filter items by name or
description...">

    <div id="itemListContainer">
        <p>Loading items...</p>
    </div>

    <script>
        let allItems = []; // Store all fetched items for client-side
filtering

        async function fetchAndDisplayItems() {
            try {
                const response = await fetch('/api/items'); // Assuming API
from Lab 9
                allItems = await response.json();
                displayItems(allItems);
            } catch (error) {
                console.error('Error fetching items:', error);
                document.getElementById('itemListContainer').innerHTML =
'<p>Error loading items.</p>';
            }
        }
```

```
        function displayItems(itemsToDisplay) {
            const container = document.getElementById('itemListContainer');
            container.innerHTML = ''; // Clear previous content

            if (itemsToDisplay.length === 0) {
                container.innerHTML = '<p>No items found.</p>';
                return;
            }

            itemsToDisplay.forEach(item => {
                const itemCard = document.createElement('div');
                itemCard.className = 'item-card';
                itemCard.innerHTML = `
                    <h3>${item.name}</h3>
                    <p>${item.description}</p>
                    <small>ID: ${item._id}</small>
                `;
                container.appendChild(itemCard);
            });
        }

        document.getElementById('filterInput').addEventListener('input',
(event) => {
            const searchTerm = event.target.value.toLowerCase();
            const filteredItems = allItems.filter(item =>
                item.name.toLowerCase().includes(searchTerm) ||
                item.description.toLowerCase().includes(searchTerm)
            );
            displayItems(filteredItems);
        });

        // Initial fetch and display when the page loads
        window.onload = fetchAndDisplayItems;
    </script>
</body>
</html>
```

Input: User typing into the filter input field.

Expected Output:

A list of items fetched from the backend API displayed on the page.

As the user types in the filter box, the displayed list dynamically updates to show only items matching the search term.

Lab 12: Making HTTP Requests from Angular to an API

Title: Consuming RESTful APIs in an Angular Application

Aim: To learn how to use Angular's `HttpClient` module to make HTTP requests to a backend API and handle responses within an Angular component.

Procedure:

1. Set up a new Angular project.
2. Import `HttpClientModule` into `app.module.ts`.
3. Inject `HttpClient` into an Angular service or component.
4. Use `HttpClient.get()`, `post()`, `put()`, `delete()` methods to interact with your backend API.
5. Subscribe to Observables returned by `HttpClient` methods to handle responses and errors.
6. Display the fetched data in an Angular component's template.

Source Code: (Conceptual Angular component and service)

```
// item.service.ts (Angular Service)
import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http';
import { Observable } from 'rxjs';

export interface Item {
  _id?: string;
  name: string;
  description: string;
}

@Injectable({
  providedIn: 'root'
})
export class ItemService {
  private apiUrl = 'http://localhost:3000/api/items'; // Your backend API URL

  constructor(private http: HttpClient) { }

  getItems(): Observable<Item[]> {
    return this.http.get<Item[]>(this.apiUrl);
  }

  addItem(item: Item): Observable<Item> {
    return this.http.post<Item>(this.apiUrl, item);
  }

  updateItem(id: string, item: Item): Observable<Item> {
    return this.http.patch<Item>(`${this.apiUrl}/${id}`, item);
  }

  deleteItem(id: string): Observable<any> {
    return this.http.delete<any>(`${this.apiUrl}/${id}`);
  }
}
```typescript
// app.component.ts (Angular Component)
import { Component, OnInit } from '@angular/core';
import { ItemService, Item } from './item.service';
```
```

```
@Component({
  selector: 'app-root',
  template: `
    <h1>Angular Items</h1>
    <div>
      <input [(ngModel)]="newItemName" placeholder="Item Name">
      <textarea [(ngModel)]="newItemDescription" placeholder="Item
Description"></textarea>
      <button (click)="addItem()">Add Item</button>
    </div>
    <ul>
      <li *ngFor="let item of items">
        {{ item.name }} - {{ item.description }}
        <button (click)="deleteItem(item._id!)">Delete</button>
      </li>
    </ul>
  `
})
export class AppComponent implements OnInit {
  items: Item[] = [];
  newItemName: string = '';
  newItemDescription: string = '';

  constructor(private itemService: ItemService) { }

  ngOnInit(): void {
    this.fetchItems();
  }

  fetchItems(): void {
    this.itemService.getItems().subscribe(
      data => this.items = data,
      error => console.error('Error fetching items:', error)
    );
  }

  addItem(): void {
    const newItem: Item = { name: this.newItemName, description:
this.newItemDescription };
    this.itemService.addItem(newItem).subscribe(
      addedItem => {
        this.items.push(addedItem);
        this.newItemName = '';
        this.newItemDescription = '';
      },
      error => console.error('Error adding item:', error)
    );
  }

  deleteItem(id: string): void {
    this.itemService.deleteItem(id).subscribe(
      () => {
        this.items = this.items.filter(item => item._id !== id);
      },
      error => console.error('Error deleting item:', error)
    );
  }
}
```

Input: User interaction with Angular UI (e.g., typing in input fields, clicking buttons).

Expected Output:

Angular application fetches and displays a list of items from the backend API.

User actions (add, delete) trigger API calls, and the Angular UI updates in real-time based on API responses.

Lab 13: Passing Data into Modal

Title: Displaying Dynamic Content in Modal Dialogs

Aim: To implement a modal dialog (popup) that displays specific data passed to it from the main application view.

Procedure:

1. Integrate a modal component library (e.g., Bootstrap Modals, Angular Material Dialogs, or a custom modal).
2. On the main view, create a list of items (e.g., from Lab 11 or 12).
3. Add a button or click event to each item that, when clicked, opens the modal.
4. Pass the specific item's data (e.g., ID, name, description) to the modal component when opening it.
5. Inside the modal, display the received data.

Source Code: (Conceptual example using Bootstrap 5 modal and plain JavaScript)

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Modal Data Passing</title>
    <link
href="[https://cdn.jsdelivr.net/npm/bootstrap@5.3.0/dist/css/bootstrap.min.cs
s](https://cdn.jsdelivr.net/npm/bootstrap@5.3.0/dist/css/bootstrap.min.css)"
rel="stylesheet">
    <style>
        .item-card { cursor: pointer; }
    </style>
</head>
<body>
    <div class="container mt-4">
        <h1>Items List</h1>
        <div id="itemListContainer" class="row">
            </div>
    </div>

    <div class="modal fade" id="itemDetailModal" tabindex="-1" aria-
labelledby="itemDetailModalLabel" aria-hidden="true">
        <div class="modal-dialog">
            <div class="modal-content">
                <div class="modal-header">
                    <h5 class="modal-title" id="itemDetailModalLabel">Item
Details</h5>
                    <button type="button" class="btn-close" data-bs-
dismiss="modal" aria-label="Close"></button>
                </div>
                <div class="modal-body">
                    <h4 id="modalItemName"></h4>
                    <p id="modalItemDescription"></p>
                    <small>ID: <span id="modalItemId"></span></small>
                </div>
                <div class="modal-footer">
                    <button type="button" class="btn btn-secondary" data-bs-
dismiss="modal">Close</button>
                </div>
```

```
            </div>
        </div>
    </div>

    <script
src="[https://cdn.jsdelivr.net/npm/bootstrap@5.3.0/dist/js/bootstrap.bundle.m
in.js](https://cdn.jsdelivr.net/npm/bootstrap@5.3.0/dist/js/bootstrap.bundle.
min.js)"></script>
    <script>
        const items = [ // Mock data, replace with API fetch in real app
            { _id: 'a1', name: 'Product A', description: 'Description for
product A.' },
            { _id: 'b2', name: 'Product B', description: 'Description for
product B.' },
            { _id: 'c3', name: 'Product C', description: 'Description for
product C.' }
        ];

        function displayItems() {
            const container = document.getElementById('itemListContainer');
            container.innerHTML = '';
            items.forEach(item => {
                const itemCard = document.createElement('div');
                itemCard.className = 'col-md-4 mb-3';
                itemCard.innerHTML = `
                    <div class="card item-card" data-bs-toggle="modal" data-
bs-target="#itemDetailModal" data-item-id="${item._id}">
                        <div class="card-body">
                            <h5 class="card-title">${item.name}</h5>
                            <p class="card-
text">${item.description.substring(0, 50)}...</p>
                        </div>
                    </div>
                `;
                container.appendChild(itemCard);
            });
        }

        // Event listener for when the modal is about to be shown
        const itemDetailModal = document.getElementById('itemDetailModal');
        itemDetailModal.addEventListener('show.bs.modal', function (event) {
            // Button that triggered the modal
            const button = event.relatedTarget;
            // Extract info from data-bs-item-id attributes
            const itemId = button.getAttribute('data-item-id');

            // Find the item data
            const item = items.find(i => i._id === itemId);

            // Update the modal's content.
            if (item) {
                document.getElementById('modalItemName').textContent =
item.name;
                document.getElementById('modalItemDescription').textContent =
item.description;
                document.getElementById('modalItemId').textContent =
item._id;
            }
        });

        window.onload = displayItems;
    </script>
</body>
</html>
```

Input: User clicking on an item card.

Expected Output: A modal dialog appears, displaying the full details (name, description, ID) of the specific item that was clicked.

Lab 14: More Complex Views and Routing Parameters

Title: Implementing Advanced UI Layouts and Dynamic Routing

Aim: To create multi-page applications with complex view structures and use routing parameters to display specific content based on the URL.

Procedure:

1. Set up a client-side routing library (e.g., Angular Router, React Router, Vue Router, or a simple custom JavaScript router).
2. Define multiple routes, including routes with parameters (e.g., `/items/:id`, `/users/:username`).
3. Create separate components/views for each route.
4. Implement navigation between routes.
5. In components/views, extract the route parameters and use them to fetch and display relevant data (e.g., fetch a single item by ID).

Source Code: (Conceptual Angular routing example)

```
// app-routing.module.ts (Angular Routing)
import { NgModule } from '@angular/core';
import { RouterModule, Routes } from '@angular/router';
import { ItemListComponent } from './item-list/item-list.component';
import { ItemDetailComponent } from './item-detail/item-detail.component';
import { PageNotFoundComponent } from './page-not-found/page-not-
found.component'; // Optional

const routes: Routes = [
  { path: 'items', component: ItemListComponent },
  { path: 'items/:id', component: ItemDetailComponent }, // Route with
parameter
  { path: '', redirectTo: '/items', pathMatch: 'full' },
  { path: '**', component: PageNotFoundComponent } // Wildcard route for 404
];

@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule { }
```typescript
// item-list.component.ts (Displays list of items)
import { Component, OnInit } from '@angular/core';
import { ItemService, Item } from '../item.service';

@Component({
  selector: 'app-item-list',
  template: `
    <h2>All Items</h2>
    <ul>
      <li *ngFor="let item of items">
        <a [routerLink]="['/items', item._id]">{{ item.name }}</a>
      </li>
    </ul>
  `
})
export class ItemListComponent implements OnInit {
  items: Item[] = [];
```
```

```
  constructor(private itemService: ItemService) { }
  ngOnInit(): void {
    this.itemService.getItems().subscribe(data => this.items = data);
  }
}
```typescript
// item-detail.component.ts (Displays single item details based on ID from
URL)
import { Component, OnInit } from '@angular/core';
import { ActivatedRoute } from '@angular/router';
import { ItemService, Item } from '../item.service';

@Component({
  selector: 'app-item-detail',
  template: `
    <div *ngIf="item">
      <h2>{{ item.name }} Details</h2>
      <p><strong>Description:</strong> {{ item.description }}</p>
      <p><strong>ID:</strong> {{ item._id }}</p>
      <button routerLink="/items">Back to List</button>
    </div>
    <div *ngIf="!item">
      <p>Item not found or loading...</p>
    </div>
  `
})
export class ItemDetailComponent implements OnInit {
  item: Item | undefined;
  constructor(
    private route: ActivatedRoute,
    private itemService: ItemService
  ) { }

  ngOnInit(): void {
    this.route.paramMap.subscribe(params => {
      const id = params.get('id');
      if (id) {
        // In a real app, you'd fetch a single item by ID from your service
        // For now, let's mock it from a list
        this.itemService.getItems().subscribe(allItems => {
          this.item = allItems.find(i => i._id === id);
        });
      }
    });
  }
}
```

Input: Navigating to different URLs (e.g., `/items`, `/items/a1`).

Expected Output:

Navigating to `/items` shows a list of all items.

Clicking on an item or navigating to `/items/:id` displays a detailed view of that specific item, with its data fetched dynamically based on the ID in the URL.
```

Lab 15: Adding and Using a Click Handler

Title: Implementing Interactive Elements with Event Handlers

Aim: To add event listeners to UI elements (e.g., buttons, links, cards) and execute specific JavaScript functions in response to user interactions.

Procedure:

1. Identify an interactive element in your UI (e.g., a "Like" button, a "Delete" icon, a "Toggle" switch).
2. Attach a click event listener to this element using JavaScript (e.g., `addEventListener` in plain JS, `(click)` in Angular, `onClick` in React).
3. Define a function that will be executed when the event occurs.
4. Inside the function, implement the desired logic (e.g., update a counter, change UI state, make an API call).

Source Code: (Example using plain JavaScript)

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Click Handler Demo</title>
    <style>
        body { font-family: sans-serif; margin: 20px; }
        #counterDisplay { font-size: 2em; margin-top: 20px; }
        .toggle-box { width: 100px; height: 100px; background-color:
lightgray; border: 1px solid #333; margin-top: 20px; cursor: pointer;
display: flex; align-items: center; justify-content: center; }
        .toggle-box.active { background-color: lightgreen; }
    </style>
</head>
<body>
    <h1>Interactive Elements</h1>

    <div>
        <button id="incrementButton">Increment Counter</button>
        <p>Current Count: <span id="counterDisplay">0</span></p>
    </div>

    <div class="toggle-box" id="toggleBox">
        Click to Toggle
    </div>

    <script>
        // Counter Example
        let count = 0;
        const counterDisplay = document.getElementById('counterDisplay');
        const incrementButton = document.getElementById('incrementButton');

        incrementButton.addEventListener('click', () => {
            count++;
            counterDisplay.textContent = count;
            console.log('Button clicked! Count:', count);
        });

        // Toggle Box Example
```

```
        const toggleBox = document.getElementById('toggleBox');
        let isActive = false;

        toggleBox.addEventListener('click', () => {
            isActive = !isActive; // Toggle the state
            if (isActive) {
                toggleBox.classList.add('active');
                toggleBox.textContent = 'Active!';
            } else {
                toggleBox.classList.remove('active');
                toggleBox.textContent = 'Click to Toggle';
            }
            console.log('Box toggled. Active:', isActive);
        });
    </script>
</body>
</html>
```

Input: User clicking the "Increment Counter" button or the "Click to Toggle" box.

Expected Output:

Clicking the "Increment Counter" button increases the displayed count by 1.

Clicking the "Click to Toggle" box changes its background color and text, alternating between active and inactive states.

Console logs confirm the click events and state changes.