ENSF-381: Full Stack Web Development Laboratory

Ahmad Abdellatif

Department of Electrical & Software Engineering

University of Calgary

Lab 4

Objectives

Welcome to the ENSF381 course lab! The objective of this lab is to provide students with practical experience in building dynamic and interactive web applications using JavaScript and browser APIs. In this lab, you will practice selecting Document Object Model (DOM) elements, attaching event handlers with addEventListener, updating webpage content and styles through the DOM, and working with an API endpoint to fetch, display, sort, and delete user data.

Groups

Lab instructions must be followed in groups of **two students**.

Constraint

Use only the JavaScript functions and data structures covered in lectures or in this lab session.

Submission

- You must submit the complete source code, ensuring it can be executed without any modifications.
- If requested by the instructor, you may need to submit additional documentation (e.g., word documents and images).
- The link to the GitHub repository created for this lab must be provided alongside your submission on D2L. If you submit a word document as a part of this lab, include the link to the GitHub repository within that document as well.
- Only one member of the group needs to submit the assignment, but the submission must include the names and UCIDS of all group members.

Deadline

Lab exercises must be submitted by **11:55 PM on the same day as the lab session**.

Exercise 1: DOM Manipulation with Event Listeners

Objective: In this exercise, we will learn to dynamically update and style elements on a webpage using the Document Object Model (DOM). Also, we will explore how to attach event handlers using the addEventListener method for better modularity and flexibility in your JavaScript code.

AddEventListener Method

We learned how to define event handlers directly in HTML attributes, such as using onkeypress or onclick. There are other methods of defining event handlers, using the addEventListener method in JavaScript. There are several benefits of using addEventListener:

1. Separation of Concerns: Keeps your HTML clean by defining the behaviour directly in your JavaScript code.
2. Multiple Handlers: Allows you to attach multiple event listeners to the same element for different events or functionalities.
3. Dynamic Addition and Removal: Enables you to dynamically add or remove event handlers as needed.

Example:

```
// Selecting an element
const button = document.getElementById('exampleButton');

// Adding a click event listener
button.addEventListener('click', () => {
    alert('Button was clicked!');
});
```

In this lab, we will apply the addEventListener method to handle keypresses for the textbox and clicks for the buttons.

Requirements

In this exercise, you need to create an interactive webpage with a textbox, a label, and five buttons. The textbox will allow users to input text. When the **Enter key** is pressed, the text will be displayed inside the label, and the textbox will be cleared. Each button should change the font color of the label to a specific color when clicked. The colours for the buttons are as follows: Red**,** Blue**,** Green**,** Orange, and Purple. You are required to use the addEventListener method to attach event handlers for both keypress and click events, and use style.color to modify the font color of the label dynamically.

Create an HTML file named <u>exercise1.html</u> with the following structure:

```
<!DOCTYPE html>

<html>
  <head>
    <title>Lab4 - Exercise 1</title>
  </head>

<body>

  <input type="text" id="textInput" placeholder="Type something here..."/>
    <label id="textLabel"></label>
       <button id="redButton" style="color:red;">Red</button>
        <button id="blueButton" style="color:blue;">Blue</button>
        <button id="greenButton" style="color:green;">Green</button>
        <button id="orangeButton" style="color:orange;">Orange</button>
       <button id="purpleButton" style="color:purple;">Purple</button>
</body>

<script type="text/javascript">

  <!-- Your JavaScript code will go here -->

</script>

</html>
```

JavaScript Implementation

Write JavaScript code in the <script> section of the HTML file using the addEventListener method to:

1. **Textbox to Label**:
    - Add an event listener to detect when the **Enter key** is pressed in the textbox.
    - Update the label with the entered text and clear the textbox.
2. **Button Clicks**:

- Add event listeners to each button.
- Change the label's font color to the corresponding color when a button is clicked. The buttons will have the following functionality:
    - Red Button: Change the label's font color to red.
    - Blue Button: Change the label's font color to blue.
    - Green Button: Change the label's font color to green.
    - Orange Button: Change the label's font color to orange.
    - Purple Button: Change the label's font color to purple.

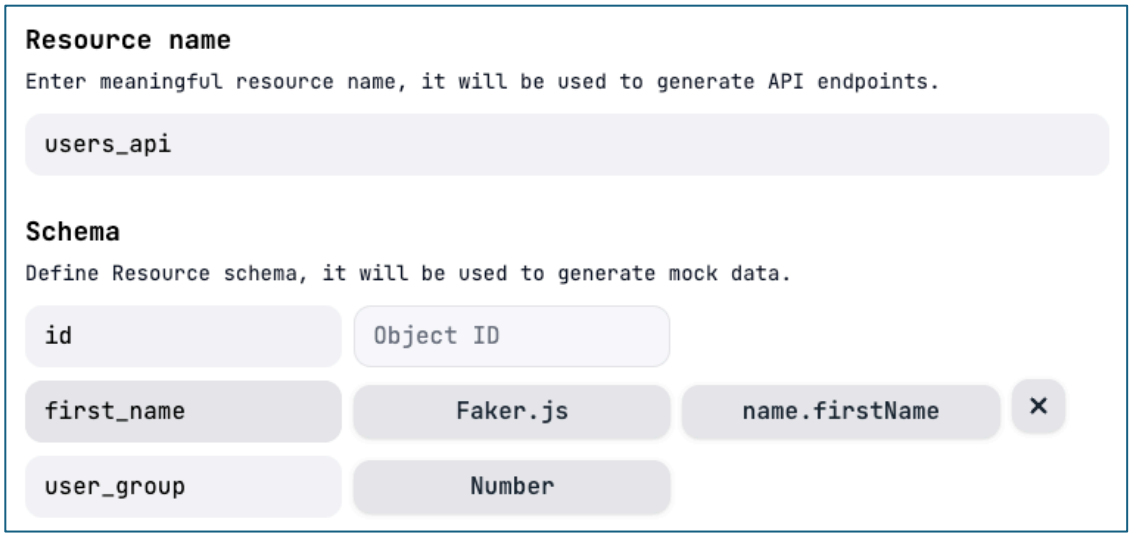
Submission

1. Fill out the Answer_sheet.docx, including:
    1. The names and UCIDs of all group members.
    2. Write the first names of all group members in the textbox, change its color to red, and include a screenshot showing the output of Exercise 1.
2. Create a new GitHub repository and upload the code for Exercise 1 to the new repository.

Exercise 2: Working with MockAPI.io

Objective: MockAPI.io is an online tool for quickly creating mock APIs. It simplifies the process of generating fake data and endpoints, making it particularly useful for prototyping applications. The objective of this exercise is to familiarize students with creating and interacting with mock APIs using MockAPI.io

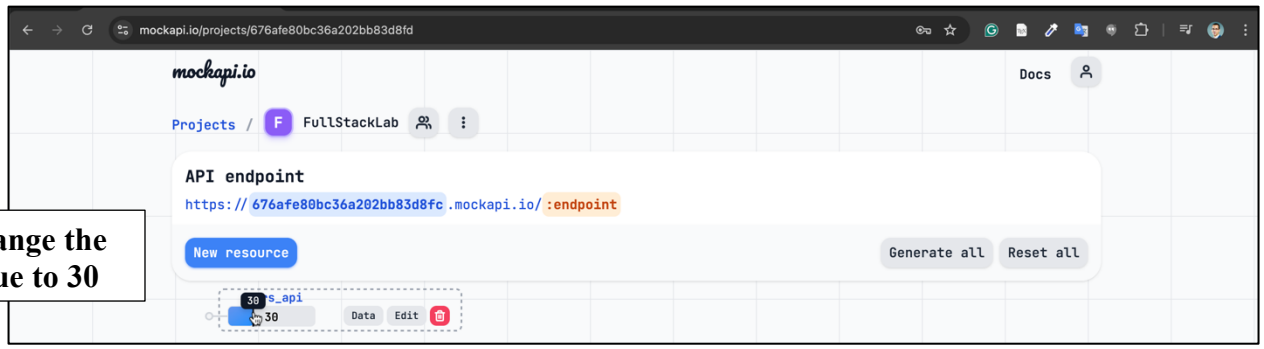
1. Open your browser and navigate to MockAPI.io: https://mockapi.io/
2. Click on "Sign Up" to create an account using your email address.
3. Log in into your account and click on "Create Project" by clicking on to create a new project.
4. Name your project: "FullStackLab".
5. Inside the FullStackLab project, click "Create resource" to add a new API.
6. Delete all fields except the 'id' field.
7. Set the API name to "users_api".
8. Add two fields for the API:
    - first_name (Faker.js) and set the random value to "name.firstName". Faker.js is a library that generates random, realistic data for testing. Using name.firstName as the random value ensures the first_name field is populated with realistic first names.
    - user_group (number type), which represents a category or group assigned to users, such as their role, type, or status. This field can be used to organize and manage users based on their role.



9. Before clicking on the ‘Create’ button, take a screenshot of the ‘Resource’ page, showing the resource name and schema that you defined. Add this screenshot to Answer_sheet.docx

10. Click on the Create button. This process creates a new resource within the project, and it initializes the API with the specified name and fields.

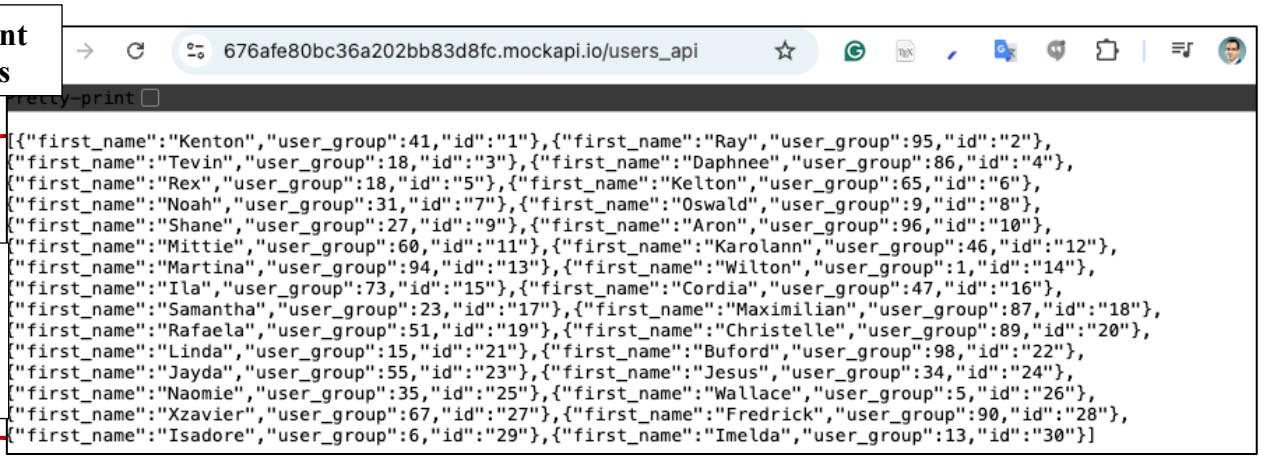
11. MockAPI.io enables us to control the number of records returned in each API response, making it ideal for simulating real-world scenarios. In this step, we will configure the API to return 30 users per request. Each time the API is called, it will consistently return a fixed number of users (i.e., 30). Click on the section that displays the number of users, as shown in the picture, and update the value to 30.



Change the value to 30

12. Take a screenshot showing the number of return items set to 30 and add it to Answer_sheet.docx.
13. After configuring the number of returned users, you can view the generated data by clicking on the API endpoint shown. The output will look similar to the following:

Endpoint Address



Returned Results

14. Take a screenshot of the page showing the generated data and add it to Answer_sheet.docx

Submission

In the Answer_sheet.docx you edited for Exercise 1, include the screenshots from steps 9, 12 and 14.

Exercise 3: Displaying Dynamic Content

Objective: In this exercise, we will practice API integration and DOM event handling by connecting the API you created in Exercise 2 to a webpage. The webpage (exercise3.html) displays users in a styled grid/list layout, but the controls are initially non-functional. In this exercise, you will use DOM selectors, addEventListener, and fetch to make the page interactive.

1. Make a copy of exercise3.html and exercise3.js. Exercise3.html contains the styling and structure of the HTML page.
2. In exercise3.js, create DOM selectors for the controls and display area. At minimum, you should select:
    - the user grid container (userGrid)
    - the view toggle button (viewToggleBtn)
    - the delete input (deleteIdInput)
    - the delete button (deleteBtn)
    - the sort by group button (sortByGroupBtn)
    - the sort by ID button (sortByIdBtn)
3. Create a users array variable to store the list returned from the API.
4. Write an **async** function retrieveData that does the following:
    1. Fetch all user data from the API (use the same endpoint address for users_api that you created in Exercise2.
    2. Store the retrieved data in the users array.
    3. log the users array to the console.

    Note: retrieveData should run as soon as the page loads.
5. Create a render function that accepts an array of user objects and uses the following string template (note the backticks) to populate the userGrid container:

```
`<article class="user-card">
  <h3>${user.first_name ?? ""}</h3>
  <p>first_name: ${user.first_name ?? ""}</p>
  <p>user_group: ${user.user_group ?? ""}</p>
  <p>id: ${user.id ?? ""}</p>
 </article>
`
```

6. Modify your retrieveData function to call the render function on the users array after it successfully fetches data from the API.

7. Add an event listener on the view toggle (*Grid / List*) button. When the button is clicked, the page should toggle between grid and list view. Implement this by modifying the class of the user grid container: if userGrid.classList contains 'grid-view', remove it and add 'list-view'; if userGrid.classList contains 'list-view', remove it and add 'grid-view'.

8. Add an event listener on the **Sort by Group** button (sortByGroupBtn). When clicked, sort the users array by user_group (ascending order), then call render(users).

9. Add an event listener on the **Sort by ID** button (sortByIdBtn). When clicked, sort the users array by the numeric value of the id (Number(user.id)) in ascending order, then call render(users).

10. Add a click event listener to the Delete button (deleteBtn). When clicked, read the user ID from the deletion input (deleteIdInput), delete that user from the API, and remove the user from the displayed list. If the ID is invalid, no matching user exists, or the API deletion request fails, log an appropriate error message to the console.

Hint: Use the fetch function with the HTTP DELETE verb to delete a user from the API. The API call should go to https://<your_api_uri>/<id_to_delete>.

11. Make sure that after deleting a user, the displayed list of users is updated to reflect the deletion. In addition, verify that after refreshing the page, the deleted user does not appear.

Submission

1. In the Answer_sheet.docx you edited for Exercise 1, include two screenshots for Exercise 3:

    a. The first screenshot should show the returned API values (the user list) displayed in the developer console.

    b. The second screenshot should show the webpage populated with user cards displaying data from the API.

2. Upload the code for Exercise 3 to the same repository you created for Exercise 1.

3. Compress both Exercise 1 and Exercise 3 into a single file and upload the compressed file to D2L. Also, upload the completed Answer_sheet.docx.