

**DUBLIN CITY UNIVERSITY**

**SCHOOL OF ELECTRONIC ENGINEERING**

 EEN1037 Web Application Development

Assignment 2

Name: Mohammed Al Shuaili

Date: 20/02/2025

### AutoGadget Hub - Web Application Briefing Document 2

**Overview:**

This document briefly describes the approach followed to accomplish Assignment 2, involving developing the client-side interface of my web application developed in assignment 1 by adding JavaScript and related methods learned in Lecture 4. The task covers four main features: creating a dynamic menu bar that responds to user interactions such as clicks or mouse movements, implementing form data validation with input field checks and user suggestions, utilizing local storage or session storage to dynamically collect and display user inputs, and handling various events such as alerts, confirms, and form submissions to trigger meaningful actions within the application.

Q1. Dynamic Menu Bar with Java Script:

The implementation of the Dynamic Menu Bar begins by waiting for the DOM content to be fully loaded using the document.addEventListener('DOMContentLoaded', function () {...}) event to ensure that the entire HTML document has been fully loaded and parsed by the browser. Once the content is ready, several elements are selected, including the side menu (sideMenu), overlay (menuOverlay), hamburger button (.hamburger), close button (.close-btn), and the dark mode toggle (#darkModeToggle).

**Toggle Menu:**

The function toggleMenu is defined to control the visibility of the menu. It toggles the active class on the menu element, which makes the menu appear or disappear. Additionally, the overlay is shown or hidden based on the presence of the active class (overlay.style.display = menu.classList.contains('active') ? 'block' : 'none';). For animation, the menu items are selected using querySelectorAll('.menu-content a'). When the menu is displayed, each menu item is animated with a staggered effect using setTimeout, modifying the opacity and transform properties to slide the items in from the left (item.style.opacity = '1'; item.style.transform = 'translateX(0)';). When the menu is closed, the items reset their appearance (item.style.opacity = '0'; item.style.transform = 'translateX(-20px)';).

**Dark Mode Toggle:**

The function toggleDarkMode manages the dark mode feature. When the dark mode toggle button is clicked, it adds or removes the dark-mode class from the body, changing the overall theme of the web application. The state is saved to localStorage so that the user's preference persists across page reloads. The dark mode button's state is also updated using darkModeToggle.classList.toggle("active", isDark);, which indicates whether dark mode is active.

**Load Dark Mode Preference:**

The function loadDarkMode is responsible for checking if the dark mode preference has been saved in localStorage. If a true value is found, dark mode is applied by adding the dark-mode class to the body element, and the toggle button is visually activated.

**Event Listeners:**

* The hamburger.addEventListener('click', toggleMenu); listens for clicks on the hamburger button to open the menu.
* The closeBtn.addEventListener('click', toggleMenu); listens for clicks on the close button to close the menu.
* The overlay.addEventListener('click', toggleMenu); ensures the menu can be closed by clicking on the overlay.
* Finally, darkModeToggle.addEventListener("click", toggleDarkMode); is used to enable or disable dark mode when the user clicks on the dark mode toggle.

By combining these features, the dynamic menu bar is made interactive, with smooth animations and the ability to toggle between light and dark themes based on the user's preference.

Q2. Form Data Validation:

Your implementation of form data validation is well-organized, and you have effectively incorporated JavaScript validation for multiple fields in both forms. Below is a breakdown of how your implementation fulfills the requirements of form validation and handles input checks:

**1. Signup Form Validation (User Registration)**

* **Username**: You check if the username is at least 3 characters long. If the validation fails, an error message is shown to the user.
* **Email**: A regular expression is used to validate the email format, ensuring that the input matches a standard email structure.
* **Password**: The password must be at least 8 characters long and contain at least one digit. If these conditions are not met, the user is informed.
* **Confirm Password**: A check is performed to ensure that the password and its confirmation match. If not, an error message is shown.
* **Address**: If the address line 1 is empty, an error message is displayed.
* **Phone**: The phone number is validated to ensure it contains exactly 10 digits, using a regular expression.

Each input field has a dedicated validation function (e.g., validateEmail, validatePassword, validatePhone) and corresponding error messages. The errors are displayed dynamically beside the relevant input fields, providing users with immediate feedback.

After the form is successfully validated, the form data is stored in localStorage, and the user entries are displayed in a table format below the form. The entries are persisted across page reloads by retrieving them from localStorage.

**2. Contact Form Validation (Message Submission)**

* **Name**: The name is checked to ensure it's at least 3 characters long.
* **Email**: Similar to the signup form, the email is validated with a regular expression.
* **Subject**: The subject must be at least 5 characters long.
* **Message**: The message should contain at least 20 characters.

Similar to the signup form, you clear the errors after each submission and validate the inputs before allowing the form to be submitted. A confirmation dialog is presented to the user before submission (confirm('Are you sure your email is correct?')), which is an extra touch to ensure the accuracy of the user's email.

Q3. Form Data Local Storage:

To meet the requirements of **Q3: Form Data Local Storage**, your code already handles user input collection and storage using **localStorage**. Here’s a detailed breakdown of how you can use localStorage to store and display form submissions dynamically, and how the button functionality works to display the responses in a table.

**How Local Storage is Used in Your Implementation:**

1. **Storing Data in localStorage:**

In both forms (the Signup and Contact forms), when the user submits the form, the data is collected, validated, and then saved to localStorage. This is done as follows:

// After validation is successful

const entry = {

formData,

timestamp: new Date().toISOString() // Adds timestamp for when the entry was made

};

// Retrieve existing entries from localStorage or initialize an empty array

const entries = JSON.parse(localStorage.getItem(SIGNUP\_STORAGE\_KEY)) || [];

// Push the new entry to the array

entries.push(entry);

// Save the updated entries back to localStorage

localStorage.setItem(SIGNUP\_STORAGE\_KEY, JSON.stringify(entries));

This code stores all form submissions in the localStorage under a unique key (SIGNUP\_STORAGE\_KEY or Contact\_STORAGE\_KEY). This allows the user’s data to persist even after the page is reloaded.

1. **Displaying Data in a Dynamic Table:**

After storing the form data, the user can click the "Show Entries" button to display all previously submitted data in a table format.

// This function loads and displays the stored entries

function loadSignupEntries() {

signupTableBody.innerHTML = ''; // Clear the table before adding new rows

const entries = JSON.parse(localStorage.getItem(SIGNUP\_STORAGE\_KEY)) || [];

entries.forEach(entry => addTableRow(entry)); // Add each entry to the table

}

// Adds a row to the table

function addTableRow(entry) {

const row = signupTableBody.insertRow();

row.innerHTML = `

<td>${entry.formData.username}</td>

<td>${entry.formData.email}</td>

<td>${entry.formData.addressLine1}${entry.formData.addressLine2 ? ', ' + entry.formData.addressLine2 : ''}</td>

<td>${entry.formData.phone}</td>

<td>${new Date(entry.timestamp).toLocaleString()}</td>

`;

}

The loadSignupEntries function retrieves all the stored data from localStorage, and for each entry, it calls addTableRow to create a new row in the table. This will display the entries dynamically when the user clicks the "Show Entries" button.

1. **Button to Display Entries:**

The "Show Entries" button is implemented as follows:

// Button to toggle visibility of the table

document.getElementById('showEntriesBtn').addEventListener('click', () => {

signupTable.style.display = signupTable.style.display === 'none' ? 'table' : 'none';

});

When the user clicks this button, it toggles the visibility of the table displaying the form entries. Initially, the table may be hidden, and upon clicking the button, the entries appear.

Q4. Event Capturing and Handling:

This question is mainly addressed in (`signup.js` and `contact.js`), three different types of events are captured and handled to enhance user interaction. Below is the breakdown:

1. Hover Events (`mouseover` and `mouseout`)

Implementation Location:

- `signup.js`: `showEntriesBtn` button (line 8–14).

- `contact.js`: `showEntriesBttn` button (line 7–13).

Functionality:

- Changes the button’s background color to “green” and text color to “white” when the user hovers over it (`mouseover`).

- Resets the button’s style when the cursor leaves (`mouseout`).

Relevance:

Provides immediate visual feedback to users, improving interactivity and guiding attention to actionable elements.

2. Submit Event (`submit`)

Implementation Location:

- `signup.js`: Form submission handler (line 46–127).

- `contact.js`: Form submission handler (line 18–72).

Functionality:

- Validates user inputs (e.g., email format, password strength, phone number).

- Stores validated data in `localStorage`.

- Dynamically updates the displayed table with new entries.

- Shows a success `alert` upon valid submission.

Relevance:

Ensures data integrity and user input correctness while enabling persistent storage and real-time UI updates.

3. Confirm Event (`confirm`)

\*\*Implementation Location\*\*:

- `contact.js`: Before form submission (line 21–23).

Functionality:

- Triggers a confirmation dialog asking, \*"Are you sure your email is correct?"\*

- Halts form submission if the user cancels the dialog.

Relevance:

Reduces user errors by prompting a final verification of critical data (email) before submission.

4. Alert event (‘alert’)