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 adding a side menu to all pages containing terms & Conditions, trending, support and a toggle dark mode. The terms and conditions button will take the user to the page containing my business policy and privacy. The trending button will direct users to the new arrival section in the shop page and support button will navigate them to the contact me form page so they can send an email to me. Finally, the dark mode toggle will change the background of the pages to dark and font to white at different levels. This menu uses a hamburger icon for opening and an ‘x’ for closing. The following shows the HTML implementation that’s added to all pages in the main body.

HTML adjustment:

```
<div class="menu-trigger">
    <div class="hamburger">☰</div>
</div>
<!-- Dynamic Side Menu -->
<nav id="sideMenu" class="side-menu">
    <div class="menu-header">
        <span class="close-btn">×</span>
        <h3>Navigation Menu</h3>
    </div>
    <div class="menu-content">
        <a href="terms.html">Terms & Conditions</a>
        <a href="shop.html#New">Trending</a>
        <a href="contact.html"> Support</a>
        <div class="setting-item">
            <span>Dark Mode</span>
            <div class="toggle-switch" id="darkModeToggle"></div>
        </div>
    </div>
</nav>
<div id="menuOverlay" class="overlay"></div>
```

Java script coding

The munu.js file implements the dynamic side menu and dark mode toggle functionality. The script runs after the DOM content is fully loaded (line 1). It starts by selecting the necessary elements: the side menu (sideMenu), overlay (menuOverlay), hamburger icon (hamburger), close button (closeBtn), and dark mode toggle (darkModeToggle) (lines 2-6).

The `toggleMenu` function (lines 9-28) handles the opening and closing of the side menu. It toggles the `active` class on the menu (`line 10`) and updates the overlay's display property (line 11). Additionally, it animates the menu items with a stagger effect, setting their opacity and transform properties based on whether the menu is active or not (lines 14-22). When the menu is closed, the items reset their appearance (lines 23-27)

The toggleDarkMode function (lines 31-38) toggles the dark-mode class on the body element (line 32), saves the user's preference in localStorage (line 34), and updates the dark mode toggle's UI state by toggling the active class (line 37).

The loadDarkMode function (lines 41-46) applies the saved dark mode preference when the page loads. It checks localStorage for the darkMode value and adds the dark-mode and active classes if dark mode was previously enabled (lines 43-44).

Event listeners are added to the hamburger icon, close button, and overlay to trigger the toggleMenu function (lines 49-51). The dark mode toggle also has an event listener to call toggleDarkMode when clicked (line 54). Finally, the loadDarkMode function is called on page load to apply the saved dark mode setting (line 56).

The following shows the results after adding the style of the menu in header_mainBody.css file.

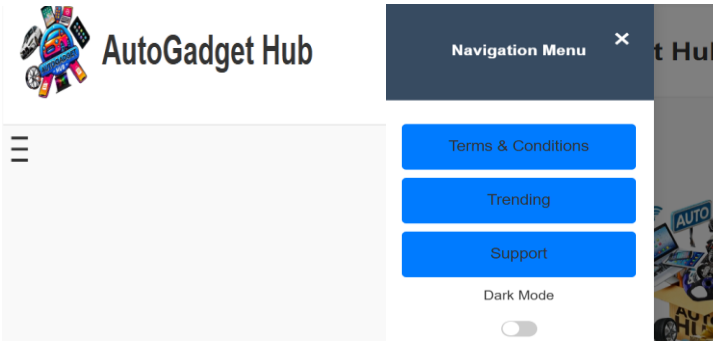

Figure 1: shows the side menu before and after opening.

The dark mode is activated by adding the .dark-mode class to the body, which changes the colour scheme to a dark theme as in header_mainBody.css file. The background is set to #1e1e1e` with white text (`#ffffff`). Specific elements like headers, side menus, links, and containers are adjusted with darker backgrounds and white text for contrast. Form inputs, tables, and product sections are styled with dark grey to maintain consistency. The footer uses a very dark background, and navigation links are set to white for readability.

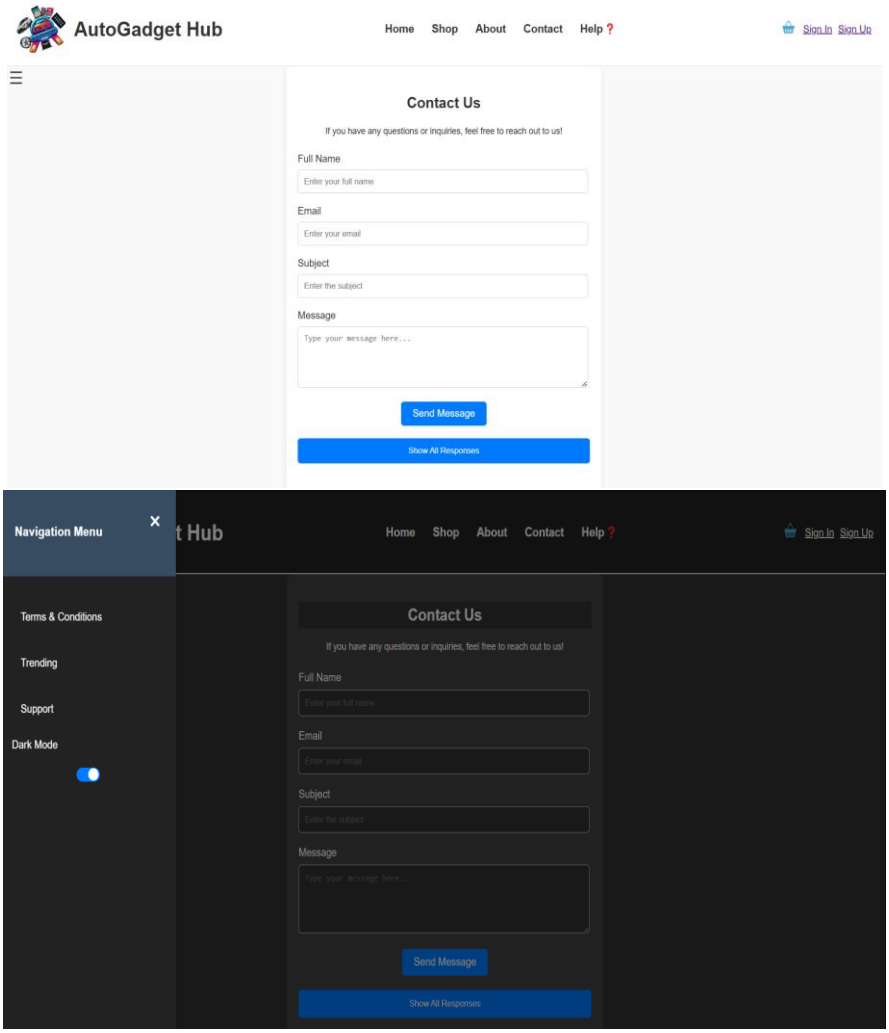

Figure 2: shows the effect of the dark mode before and after toggling.

Q2. Form Data Validation:

The implementation of this question is added into 2 main forms for multiple fields in both forms (signup.html and contact.html). Below is a breakdown of how it's done to fulfil the requirements of form validation and handles input checks:

The validation in the signup form is implemented through a series of functions that check the input values for correctness and display error messages if any validation fails.

1. Signup Form Validation (User Registration)

- **Username**: a check if the username is at least 3 characters long. If the validation fails, an error message is shown to the user (lines 84-87).
- **Email**: A regular expression that's adapted from [1] is used to validate the email format, ensuring that the input matches a standard email structure (lines 19-22).

- **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 (lines 24-26).
- **Confirm Password**: A check is performed to ensure that the password and its confirmation match. If not, an error message is shown (lines 99-101).
- **Address**: If the address line 1 is empty, an error message is displayed (lines 104-107).
- **Phone**: The phone number is validated to ensure it contains exactly 10 digits, using a regular expression (lines 28-31).

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 as shown below:

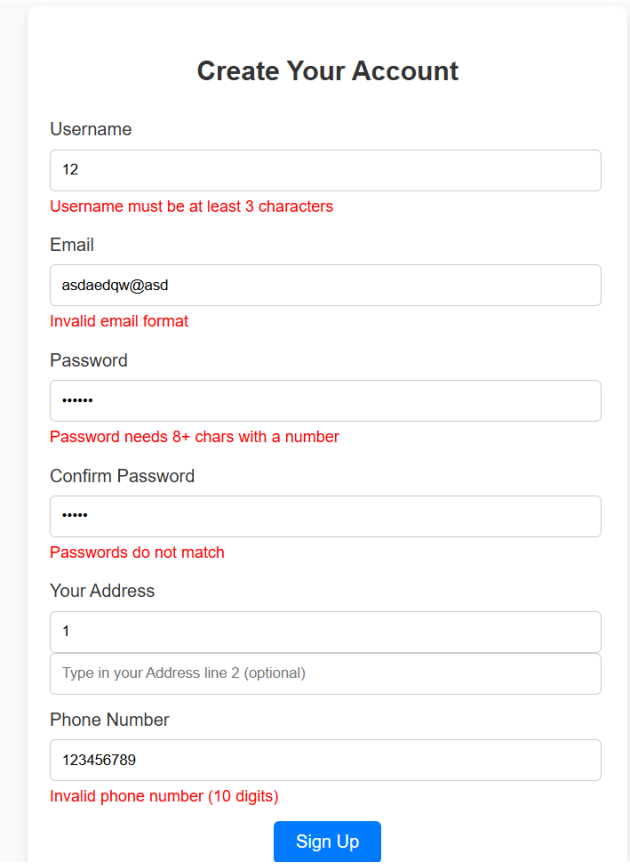

Figure 3: demonstrates the effect of the validation implementation in the signup form.

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 Q3. The entries are persisted across page reloads by retrieving them from localStorage.

2. Contact Form Validation

The same method is followed here except that in signup form the validation is divided into a series of functions while here there is no need for that contact.js.

- **Name**: The name is checked to ensure it's at least 3 characters long (lines 35-38).
- **Email**: Similar to the signup form, the email is validated with a regular expression (lines 41-44).
- **Subject**: The subject must be at least 5 characters long (lines 47-50).
- **Message**: The message should contain at least 20 characters (lines 53-56).

Similar to the signup form, the errors are cleared after each submission and validate the inputs before allowing the form to be submitted.

Q3. Form Data Local Storage:

This question is addressed by handling user input collection and storage using localStorage using the same file codes as in the previous section (signup.js and contact.js). Here's a breakdown of how the localStorage is used to store and display form submissions dynamically, and how the button functionality works to display the responses in a table.

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.

2. 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. In addition, it checks if the user added an optional field (i.e., address line 2) and adds it to the table if so.

3. 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 once clicked it shows the table and vis versa. The table is hidden by default using the style="display: none;" attribute in the html file.

Previous messages

Name	Email	Subject	Message	Date
Mohammed Alshuaili	mohadmnssnm@gmail.com	this is a test example	Hello there how are how , could you help me order ??	2/26/2025, 3:18:37 PM
mohad salim	mohadmnssnm@gmail.com	request an item	hi, are you planning to sell video games in the future?	2/26/2025, 3:19:33 PM
Ahmed	mohammed.alshuaili2@mail.dcu.ie	BMW paint request	Hey, do you sell BMW paint (B06) for f30 2016?	2/26/2025, 3:20:40 PM
Salim	mohadmnssnm@gmail.com	Help_me_with_FYP	Hey, do you know the meaning of Pneumonoultramicroscopicsilicovolcanoconiosis	2/26/2025, 3:23:01 PM

Figure 4: shows an example of stored data in a table, displaying the information submitted by users through the contact form.

Q4. Event Capturing and Handling:

This question is mainly addressed in (`signup.js` and `contact.js`), four main different types of events are captured and handled. Below is the breakdown:

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

Implementation Location:

- `signup.js`: `showEntriesBtn` button (line 8–16).

- `contact.js`: `showEntriesBttn` button (line 7–16).

This changes the button's background colour to green and text colour to white when the user hovers over it (`mouseover`) and it resets the button's style when the cursor leaves (`mouseout`).

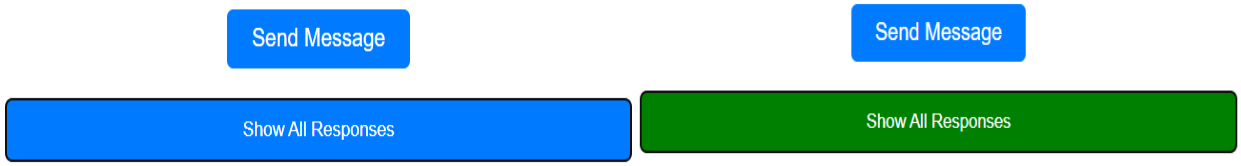

Figure 5: illustrates the use of hover events (mouseover and mouseout) before and after hovering.

Relevance:

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

2. Submit Event (`submit`)

Implementation Location:

- `signup.js`: Form submission handler (line 66–130).

- `contact.js`: Form submission handler (line 18–74).

This validates user inputs (e.g., email format, password strength, phone number). Then stores validated data in `localStorage` and dynamically updates the displayed table with new entries and at the end it shows a success `alert` upon valid submission.

Relevance:

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

3. Confirm Event (`confirm`)

Implementation Location:

- `contact.js`: Before form submission (line 23–25).

This triggers a confirmation dialog asking, "Are you sure your email is correct?" and stops form submission if the user cancels the dialog as shown below:

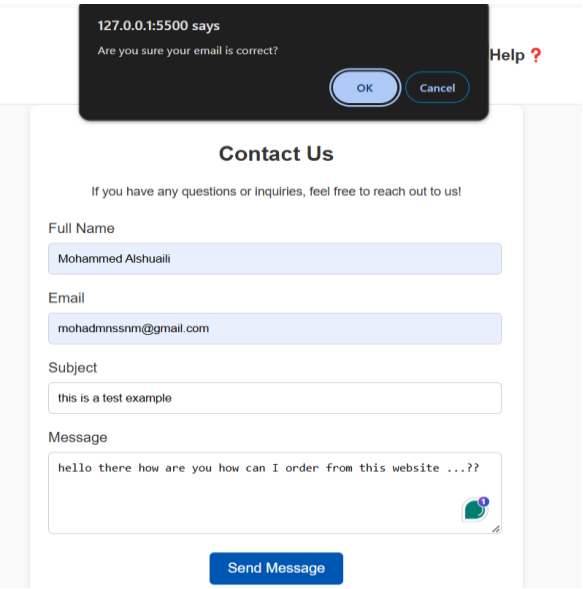

Figure 6: shows the confirm event, which is triggered when the user attempts to submit the contact form.

Relevance:

Prompting a final verification of critical data (email) before submission to ensure we contact the right person and show some carness to their concerns.

4. Alert event (‘alert’)

Implementation Location:

- `signup.js`: After submission success (line 128).

- `contact.js`: After submission success (line 71).

This alert event is triggered to inform the user that the form has been submitted successfully, and all data has been stored.

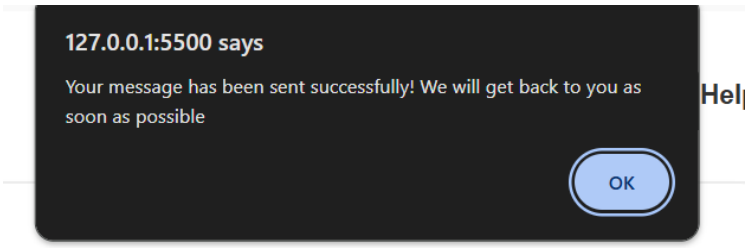

Figure 7: shows the alert event, that is triggered when the user submits the contact form.

Relevance:

It provides immediate feedback to the user, confirming that their input has been processed correctly.

Reference:

[1] "How can I validate an email address in JavaScript?", *Stack Overflow*, Sep. 11, 2008. [Online]. Available: https://stackoverflow.com/questions/46155/how-can-i-validate-an-email-address-in-javascript. [Accessed: Feb. 26, 2025].