Learning

1875 points

Front-End Web Development
Average: 83.84%

Week 6

Javascript Deep Dive

- Advanced JavaScript Features
- Asynchronous JavaScript and APIs
- Forms and Input Validation
- How Scoring Works
- How PLDs Work (Step-by-Step) - Get it started!
- Deep Dive into JavaScript
- Advanced Application of Javascript

Deep Dive into JavaScript

Novice

Weight: 1

Ongoing second chance project - started Jul 28, 2025 12:00 AM, must end by Aug 11, 2025 12:00 AM

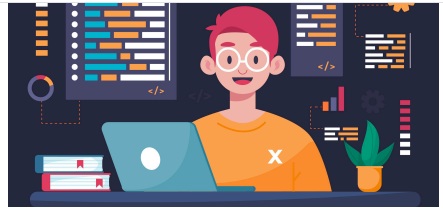
An auto review will be launched at the deadline

In a nutshell...

- **Auto QA review:** 0.0/49 mandatory
- **Altogether: 0.0%**
  - Mandatory: 0.0%
  - Optional: no optional tasks

Learning

1875 points



Welcome to the "Deep Dive into JavaScript" project! In this project, you will develop a user registration form with validation and fetch user data from a public API. This project will enhance your understanding of form validation, DOM manipulation, and asynchronous JavaScript.

You will learn how to:

- Validate form inputs using JavaScript.
- Dynamically interact with HTML elements.
- Fetch and display data from a public API.
- Provide real-time feedback to users.

By the end of this project, you will have a solid understanding of advanced JavaScript concepts and be able to create dynamic and interactive web applications.

Learning Objectives

By the end of this project, students should be able to:

1. **Implement Form Validation**:
   - Understand and implement basic form validation using JavaScript.
   - Ensure user inputs meet specified criteria before form submission.
2. **Work with the DOM**:
   - Use DOM manipulation to dynamically interact with HTML elements.
   - Understand and utilize event listeners for form submission and input validation.
3. **Asynchronous JavaScript and APIs**:
   - Use JavaScript to fetch data asynchronously from a public API.
   - Display fetched data dynamically on a webpage.
   - Handle potential errors during data fetching.

form validation.
- Ensure data persists across browser sessions using local storage.

Tasks

0. Form Creation and Validation **mandatory**

Score: 0.0% (*Checks con*

Repo Requirements :

- Create a New Repo in Github
- Repo Name : `Form-Creation-Validation`

Using this user registration form, implement validations to ensure that users provide the right kind of data.

HTML Structure:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>User Registration Form</title>
    <link rel="stylesheet" href="style.css">
</head>
```



```
</label>
        <input type="text" id="username"
required>

        <label for="email">Email:</label>
        <input type="email" id="email"
required>

        <label for="password">Password:
</label>
        <input type="password" id="password"
required>

        <button
type="submit">Register</button>
        <div id="form-feedback"></div>
    </form>
    <script src="script.js"></script>
</body>
</html>
```

CSS (style.css):

```
body {
    font-family: 'Arial', sans-serif;
    display: flex;
    justify-content: center;
    align-items: center;
    height: 100vh;
    background-color: #f5f5f5;
    margin: 0;
}

form {
    background: #ffffff;
    padding: 20px;
    border-radius: 8px;
    box-shadow: 0 4px 6px rgba(0, 0, 0,
0.1);
    width: 100%;
    max-width: 400px;
}

label {
    margin-bottom: 5px;
    font-weight: bold;
    color: #333;
}

input {
    padding: 10px;
    margin-bottom: 20px;
    border: 1px solid #ccc;
    border-radius: 4px;
    width: calc(100% - 22px); /* Adjust
width to account for padding and border */
    box-sizing: border-box; /* Include
padding and border in element's total width
```

```
button {
    background-color: #007bff;
    color: white;
    padding: 10px 15px;
    border: none;
    border-radius: 4px;
    font-size: 16px;
    cursor: pointer;
    width: 100%;
    box-sizing: border-box;
    transition: background-color 0.3s ease;
}

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

#form-feedback {
    margin-top: 10px;
    padding: 10px;
    color: #d8000c;
    background-color: #ffbaba;
    border-radius: 4px;
    display: none; /* Initially hide the
feedback div */
}
```

Implement a form validation script using basic JavaScript string methods and conditions. Upon form submission, validate the input fields for `username`, `email`, and `password` according to specific criteria. Display a success message if all validations pass, or appropriate error messages if any validations fail.

Task Requirements

Setup and Initial Code Structure

1. **Start with DOMContentLoaded Event**:
   - Wrap your entire script in a `DOMContentLoaded` event listener. This ensures your JavaScript runs after the entire HTML document has been loaded.
2. **Form Selection**:
   - Use `document.getElementById` to select the form with `id="registration-form"`. Store this reference in a constant named `form`.
3. **Feedback Division Selection**:
   - Similarly, select the division where feedback will be displayed (`id="form-

Form Submission and Event Prevention

1. **Form Submission Event Listener**:
   - Add an event listener to `form` for the 'submit' event. Use an anonymous function to handle the event.
   - Inside this function, call `event.preventDefault()` to prevent the form from submitting to the server. This is crucial for client-side validation.

Input Retrieval and Trimming

1. **Retrieve User Inputs**:
   - Use `document.getElementById` to select each input field by its respective ID: `username`, `email`, and `password`.
   - For each, retrieve the `.value` property and apply the `.trim()` method to remove any leading or trailing whitespace. Store these trimmed values in constants named after each input field.

Validation Logic

1. **Initialize Validation Variables**:
   - Declare a variable named `isValid` and set it to `true`. This will track the overall validation status.
   - Declare an array named `messages` to store validation error messages.
2. **Username Validation**:
   - Check if `username.length` is less than 3. If so, set `isValid` to `false` and add a specific error message to `messages`.
3. **Email Validation**:
   - Check if `email` includes both '@' and '.' characters. If not, set `isValid` to `false` and append a corresponding error message to `messages`.
4. **Password Validation**:
   - Ensure `password.length` is at least 8. If it falls short, update `isValid` to `false` and add an appropriate error message to `messages`.

Displaying Feedback

1. **Feedback Display Logic**:

Learning

1875 points

textContent of feedbackDiv to "Registration successful!" and its style.color to "#28a745".

- If isValid is false, join messages with <br> to form a single string, and assign this to the innerHTML of feedbackDiv. Set feedbackDiv.style.color to "#dc3545".

Repo:

- GitHub repository: Form-Creation-Validation
- File: index.html, style.css, script.js

Check submission | Get a sandbox | View results

1. Implement a Simple Interactive Quiz

mandatory

Score: 0.0% (Checks con

Requirement

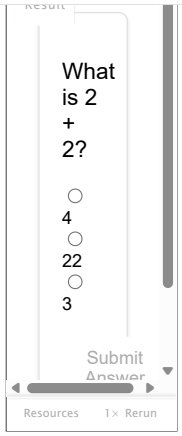
- Create a Github Repo with a name ALX_Simple_Quiz
- Make sure to start the repo with files index.html and styles.css on the same directory

Your task is to implement the JavaScript functionality for a quiz application

You will be working with the provided HTML and CSS template, which structures and styles the quiz. Your goal is to bring this quiz to life, making it interactive and functional.

Learning

1875 points



Here is the HTML and CSS code that you should use

HTML Code

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Simple Quiz</title>
</head>
<body>
    <div id="quiz-container">
        <p id="quiz-question">What is 2 + 2?
</p>
        <div>
            <input type="radio" id="choice1"
name="quiz" value="4">
            <label for="choice1">4</label>
<br>
            <input type="radio" id="choice2"
name="quiz" value="22">
            <label for="choice2">22</label>
<br>
            <input type="radio" id="choice3"
name="quiz" value="3">
            <label for="choice3">3</label>
        </div>
        <button id="submit-answer">Submit
Answer</button>
        <p id="feedback"></p>
    </div>
    <script src="quiz.js"></script>
</body>
</html>
```

CSS Code

```
    padding: 10px;
}

#quiz-container {
    max-width: 600px;
    margin: auto;
    padding: 20px;
    border: 1px solid #ddd;
    border-radius: 8px;
    box-shadow: 0 2px 4px rgba(0,0,0,0.1);
}

#quiz-question {
    font-size: 20px;
    margin-bottom: 20px;
}

input[type="radio"] {
    margin-right: 10px;
}

label {
    margin-right: 30px;
    cursor: pointer;
}

#submit-answer {
    display: block;
    margin-top: 20px;
    padding: 10px 20px;
    background-color: #007bff;
    color: white;
    border: none;
    border-radius: 5px;
    cursor: pointer;
    font-size: 16px;
}

#submit-answer:hover {
    background-color: #0056b3;
}

#feedback {
    margin-top: 20px;
    font-size: 18px;
}
```

Given the final expected JavaScript output for the Simple Interactive Quiz task, let's create a set of explicit instructions. These will guide the learners to produce the exact code, ensuring uniform submissions suitable for automated checking.

Your Task:

displays appropriate feedback. You will also add an event listener to the "Submit Answer" button to invoke this function when clicked.

Detailed Instructions:

1. **Start with the Function Declaration**:
   - Define a function named `checkAnswer`. This function will be called when the user submits their answer.

```
function checkAnswer() {
    // Function body
}
```

1. **Identify the Correct Answer**:
   - Inside the `checkAnswer` function, declare a variable named `correctAnswer` and assign it the string value `"4"`. This represents the correct answer to your quiz question.
2. **Retrieve the User's Answer**:
   - Use `document.querySelector` to select the checked radio button by its name attribute `name="quiz"`. Since radio inputs are used for the answers, this will effectively capture the user's selection.
   - Access the `value` property of the selected radio button to get the user's answer. Store this value in a variable named `userAnswer`.
3. **Compare the User's Answer with the Correct Answer**:
   - Implement an `if` statement to compare `userAnswer` with `correctAnswer`.
     - If the values match, indicating the user's answer is correct, change the `textContent` of the element with the ID `feedback` to `"Correct! Well done."`.
     - If the values do not match, indicating the user's answer is incorrect, update the `textContent` of the `feedback` element to `"That's incorrect. Try again!"`.
4. **Add an Event Listener to the Submit Button**:
   - Use `document.getElementById` to select the "Submit Answer" button by its ID,

the callback to be executed when the button is clicked. Ensure you are not calling the `checkAnswer` function directly in the event listener (i.e., do not add `()` after the function name in the `addEventListener` call).

Repo:

- GitHub repository: `ALX_Simple_Quiz`
- File: `styles.css, index.html, quiz.js`

Check submission | Get a sandbox | View results

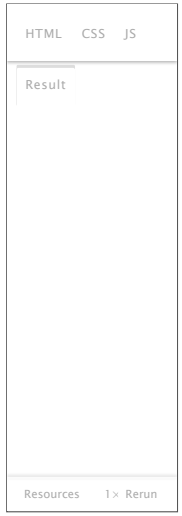
2. Build a Simple Calculator

mandatory

Score: 0.0% (*Checks con*

For this task, let's structure a simple calculator application. This application will perform basic arithmetic operations such as addition, subtraction, multiplication, and division.

This task will help you apply fundamental JavaScript concepts to create an interactive web application.



for the numbers and buttons for the operations.

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Simple Calculator</title>
    <link rel="stylesheet"
href="calculator.css">
</head>
<body>
    <div id="calculator-container">
        <input type="number" id="number1"
placeholder="Number 1">
        <input type="number" id="number2"
placeholder="Number 2">

        <button id="add">+</button>
        <button id="subtract">-</button>
        <button id="multiply">*</button>
        <button id="divide">/</button>

        <div id="result">Result: <span
id="calculation-result">0</span></div>
    </div>
    <script src="calculator.js"></script>
</body>
</html>
```

CSS Code

Style the calculator to improve its appearance and usability.

```
body {
    font-family: Arial, sans-serif;
    display: flex;
    justify-content: center;
    align-items: center;
    height: 100vh;
    margin: 0;
}

#calculator-container {
    text-align: center;
}

input[type="number"] {
    margin: 10px;
    padding: 10px;
    width: 200px;
    font-size: 16px;
}

button {
    padding: 10px 20px;
    margin: 5px;
```

```
#result {
    margin-top: 20px;
    font-size: 20px;
}
```

JavaScript Task Instructions

Objective: Implement the JavaScript to make the calculator operational. Each button should perform its respective arithmetic operation on the two input numbers and display the result.

JavaScript Implementation:

1. **Implement Arithmetic Functions:** Each arithmetic operation (add, subtract, multiply, divide) should have its own function. For example:

```
function add(number1, number2) {
    return number1 + number2;
}
```

Implement similar functions for subtraction, multiplication, and division.

1. **Attach Event Listeners:** For each operation button, add an event listener that calls the corresponding arithmetic function when clicked. Use the input values from the number fields as arguments for these functions. Display the result in the `#calculation-result` span.

Example for the addition button:

```
document.getElementById('add').addEventListe
ner('click', function() {
    const number1 =
parseFloat(document.getElementById('number1'
).value) || 0;
    const number2 =
parseFloat(document.getElementById('number2'
).value) || 0;
    const result = add(number1, number2);
    document.getElementById('calculation-
result').textContent = result;
});
```

Repeat similar steps to attach event listeners for the subtract, multiply, and divide buttons.

default values to handle empty inputs.

Repo:

- GitHub repository: `ALX_Simple_Quiz`
- File: `calculator.html, calculator.css, calculator.js`

Check submission | View results

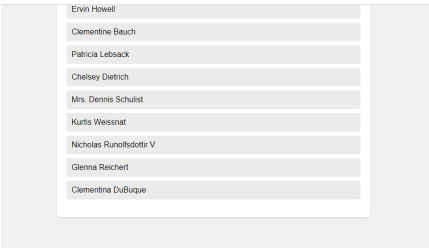
3. Fetching Data from an API and Displaying It **mandatory**

Score: 0.0% (*Checks cor*

Implement JavaScript functionality to asynchronously fetch user data from a public API (https://jsonplaceholder.typicode.com/users) and display the names of the users in a list on the webpage.

Final output

After implementing the JavaScript code, you should end up with something like this:



HTML Code

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>User Data Display</title>
    <link rel="stylesheet" href="fetch-
data.css">
</head>
<body>
    <div id="content">
        <h1>User List</h1>
        <div id="api-data">Loading user
data...</div>
    </div>
    <script src="fetch-data.js"></script>
</body>
</html>
```

CSS Code

```
body {
    font-family: Arial, sans-serif;
    margin: 0;
    padding: 20px;
    background-color: #f4f4f4;
}

#content {
    max-width: 600px;
    margin: auto;
    background: white;
    padding: 20px;
    border-radius: 8px;
    box-shadow: 0 2px 4px rgba(0,0,0,0.1);
}

h1 {
    text-align: center;
}

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



```
    background-color: #eee;
    margin: 8px 0;
    padding: 10px;
    border-radius: 4px;
}
```

Detailed Instructions for JavaScript Task

Overview

You will write JavaScript code to asynchronously fetch user data from a public API and display each user's name on a web page. Your code will also handle potential errors during the fetch operation.

JavaScript Code Writing Steps

1. **Initialize the Async Function**:
    - Begin by defining an asynchronous function named `fetchUserData`. This function will contain all your code for fetching and displaying the data.
2. **Define the API URL**:
    - Inside `fetchUserData`, declare a constant `apiUrl` and assign it the string value `'https://jsonplaceholder.typicode.com/users'`. This URL points to the API endpoint from which you'll fetch user data.
3. **Select the Data Container Element**:
    - Select the HTML element where the API data will be displayed by using `document.getElementById`. Look for an element with the ID `'api-data'` and store this reference in a constant named `dataContainer`.
4. **Fetch Data Using `try-catch`**:
    - Employ a `try-catch` block to handle the fetching process and potential errors.
    - In the `try` block, use the `await` keyword with the `fetch` function to asynchronously get data from `apiUrl`. Store the response in a constant named `response`.
    - Then, convert the response to JSON using `await response.json()` and store this data in a constant named `users`.
5. **Clear the Loading Message**:

setting `dataContainer.innerHTML = ''`.

6. **Create and Append User List**:
   - Create a `<ul>` element and store it in a constant named `userList`.
   - Loop through the `users` array with `forEach`, and for each user, do the following:
     - Create a `<li>` element.
     - Set the text content of the `<li>` to the user's name.
     - Append the `<li>` to `userList`.
   - After the loop, append `userList` to `dataContainer`.
7. **Error Handling**:
   - In the `catch` block, clear the existing content of `dataContainer` (similar to step 5) and set its text content to `'Failed to load user data.'`, indicating an error occurred during data fetching.
8. **Invoke `fetchUserData` on DOMContentLoaded**:
   - Outside `fetchUserData`, add an event listener to `document` for the `DOMContentLoaded` event. Set the callback function to invoke `fetchUserData`. This ensures your data fetching logic runs once the HTML document has fully loaded.

Repo:

- GitHub repository: `Form-Creation-Validation`
- File: `fetch-data.html, fetch-data.css, fetch-data.js`

Check submission | View results

« Back | Next »