**Tasks:**

1.      Answer to following questions (2 pts):

**a.      Compare different HTTP *methods*? Explain also how they should be used.**

**GET:**

Use: Retrieve data from a specified resource.

Idempotent: Yes (repeated requests have the same effect as a single request).

Should not have a body in the request.

**POST:**

Use: Submit data to be processed to a specified resource.

Not Idempotent: No (repeated requests may have different effects).

Used when creating a new resource or performing a non-idempotent operation.

**PUT:**

Use: Update a resource or create it if it doesn't exist at the specified URI.

Idempotent: Yes.

Replaces the entire resource at the specified URI.

**PATCHES:**

Use: Apply partial modifications to a resource.

Idempotent: Well.

Used when updating part of a resource.

**DELETE:**

Use: Delete a specified resource.

Idempotent: Yes.

Deletes the resource at the specified URI.

**OPTIONS:**

Use: Retrieve the communication options for the target resource.

Idempotent: Yes.

Used to describe the communication options for the target resource.

**HEAD:**

Use: Same as GET, but only retrieves the headers, not the actual data.

Idempotent: Yes.

Used when we want to retrieve metadata about a resource without the body content.

**b.      You have seen the HTTP response code 404. What it means and what are the possible reasons for that?**

Meaning: HTTP status code 404 indicates that the server did not find the requested resource.

Possible Reasons:

The URL or URI is incorrect.

The resource has been moved or deleted.

There is a typo in the URL.

The resource is temporarily unavailable.

The server or website is misconfigured.

Permissions or authentication issues.

**c.      Explain HTTP sandboxing**

HTTP Sandboxing:

Sandboxing in the context of HTTP typically refers to restricting or isolating the execution of web content within a controlled environment for security purposes.

**Content Security Policy (CSP):** It's a mechanism to mitigate the risk of Cross-Site Scripting (XSS) attacks by allowing or blocking content based on a set of rules.

**Cross-Origin Resource Sharing (CORS):** Defines how web pages can request and manage resources from other domains. It helps in preventing unauthorized access to resources.

**Iframes and Same-Origin Policy:** Browsers enforce the Same-Origin Policy, restricting web pages from making requests to a different domain than the one that served the web page. Iframes are often used to sandbox content within a separate frame.

**HTTP Cookies and Secure Flag:** Cookies can be marked with the "Secure" flag to ensure they are only sent over HTTPS connections, Enhancing security.

**HTTP Strict Transport Security (HSTS):** A policy mechanism to protect websites against man-in-the-middle attacks such as protocol downgrade attacks and cookie hijacking.

Sandboxing techniques are crucial for Enhancing the security of web applications and protecting users from various online threats.

2.      Create a form for data collection. Form should have inputs for: (2 pts)

a.      User contact information

b.      5 questions where user chooses numeric value between 1-3 using radio input (hint: grouping)

c.      Text area for free feedback

<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

    <meta name="viewport" content="width=device-width, initial-scale=1.0">

    <title>Data Collection Form</title>

    <style>

        label {

            display: block;

            margin-top: 10px;

        }

    </style>

</head>

<body>

    <form>

        <!-- User Contact Information -->

        <label for="fullName">Full Name:</label>

        <input type="text" id="fullName" name="fullName" required>

        <label for="email">Email:</label>

        <input type="email" id="email" name="email" required>

        <label for="phone">Phone Number:</label>

        <input type="tel" id="phone" name="phone" required>

        <!-- 5 Questions with Numeric Values -->

        <fieldset>

            <legend>Choose a numeric value between 1-3:</legend>

            <label>Question 1: <input type="radio" name="q1" value="1" required> 1</label>

            <label><input type="radio" name="q1" value="2" required> 2</label>

            <label><input type="radio" name="q1" value="3" required> 3</label>

            <!-- Repeat similar structure for Questions 2-5 -->

        </fieldset>

        <!-- Text Area for Free Feedback -->

        <label for="feedback">Feedback:</label>

        <textarea id="feedback" name="feedback" rows="4" required></textarea>

        <!-- Submit Button -->

        <button type="submit">Submit</button>

    </form>

</body>

</html>

A screenshot of a computer

Description automatically generated

**3.      Explain how you prevent a form submission using JavaScript and why would you do that? Give also a simple code example (2 pts).**

**Preventing Form Submission using JavaScript:**

In JavaScript, you can prevent the default behavior of a form, which is to submit and Reload the page, by attaching an event listener to the form's submit event and calling the preventDefault() method on the event object. This is useful in scenarios where you want to validate the form data using JavaScript before allowing the submission.

**Why We Would Do That:**

*Client-Side Validation:* You might want to perform client-side validation to ensure that the form data is correct and meets certain criteria before sending it to the server.

*Asynchronous Operations:* If the form Submission involves Asynchronous operations (e.g., AJAX requests), you may want to prevent the default form Submission behavior and handle the data asynchronously without Reloading the entire page.

Code Example:

<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

    <meta name="viewport" content="width=device-width, initial-scale=1.0">

    <title>Prevent Form Submission Example</title>

</head>

<body>

    <form id="myForm">

        <label for="username">Username:</label>

        <input type="text" id="username" name="username" required>

        <label for="password">Password:</label>

        <input type="password" id="password" name="password" required>

        <button type="submit">Submit</button>

    </form>

    <script>

        document.getElementById('myForm').addEventListener('submit', function(event) {

            // Example validation logic

            var username = document.getElementById('username').value;

            var password = document.getElementById('password').value;

            if (username === '' || password === '') {

                // Validation fails

                event.preventDefault();

                alert('Please fill out both username and password fields.');

            }

        });

    </script>

</body>

</html>

A screenshot of a computer

Description automatically generated

In this example, the validation logic checks whether the username and password fields are both non-empty. If either of them is empty, the validation fails, and event.preventDefault() is called to prevent the default form submission. An alert is then shown to notify the user about the validation failure. You should adapt the validation logic according to your specific requirements.

**4.      What means local storage related to JavaScript? How could you use it? Write an example code that (2 pts)**

**a.      has one input for text**

**b.      on blur-event writes contents in the local storage**

**c.      another code reads it when window is loaded**

**d.      Demonstrate by refreshing window…**

**Local Storage in JavaScript:**

Local Storage is a web storage solution provided by browsers, allowing web applications to store key/value pairs persistently on a user's device. It provides a simple way to save data that persists even after the user closes and reopens the browser or refreshes the page.

<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

    <meta name="viewport" content="width=device-width, initial-scale=1.0">

    <title>Local Storage Example</title>

</head>

<body>

    <label for="textInput">Enter Text:</label>

    <input type="text" id="textInput">

    <script>

        // Event listener for blur event on the input

        document.getElementById('textInput').addEventListener('blur', function () {

            // Get the input value

            var inputValue = document.getElementById('textInput').value;

            // Save the input value to local storage

            localStorage.setItem('savedText', inputValue);

        });

        // Function to read from local storage when the window is loaded

        window.onload = function () {

            // Get the saved value from local storage

            var savedText = localStorage.getItem('savedText');

            // Display the saved value

            if (savedText) {

                document.getElementById('textInput').value = savedText;

            }

        };

    </script>

</body>

</html>

Explanation:

The HTML file includes an input element for text entry.

The JavaScript code adds an event listener to the input for the blur event (when the input loses focus). On blur, it saves the input value to local storage using localStorage.setItem.

When the window is loaded (window.onload), another function retrieves the saved value from local storage using localStorage.getItem and sets the input value accordingly.

When you refresh the window, the saved text is retrieved from local storage, and the input field is populated with the saved value.

This example demonstrates a basic use case of local storage to persist data between page loads or refreshes.

5.      Make exercise “Content negotiation” from book (Exercises, chapter 18).  (2 pts)

// Function to fetch resource with a specified Accept header

async function fetchResource(mediaType) {

    const url = 'https://eloquentjavascript.net/author';

    try {

        const response = await fetch(url, {

            headers: {

                'Accept': mediaType

            }

        });

        if (response.ok) {

            const data = await response.text();

            console.log(`Success! Content for ${mediaType}:\n`, data);

        } else {

            console.log(`Error! Status Code: ${response.status}`);

        }

    } catch (error) {

        console.error('Error:', error);

    }

}

// Fetch different formats: plaintext, HTML, JSON

fetchResource('text/plain');

fetchResource('text/html');

fetchResource('application/json');

// Try asking for a non-standard media type

fetchResource('application/rainbows+unicorns');

Explanation:

The fetchResource function takes a media type as an argument and sends a fetch request with the specified Accept header.

It uses the await keyword to wait for the response, and then checks if the response is successful (status code 2xx). If successful, it logs the content; otherwise, it logs the error status code.

The example fetches the resource in three different formats: plaintext, HTML, and JSON.

It also attempts to fetch the resource with the non-standard media type 'application/rainbows+unicorns' to observe the response status code.