**What is javascript?**

JavaScript is a **high-level**, **interpreted** programming language commonly used to create and control **dynamic website content.**

Alongside HTML and CSS, it is one of the core technologies of the World Wide Web.

Here are some key aspects of JavaScript:

1. **Client-Side Scripting:**
   1. JavaScript is primarily used for client-side scripting, allowing developers to create interactive web pages.
   2. This means it runs in the user's browser and can respond to user actions like clicks, form submissions, and other events.
2. **Dynamic Content:**
   1. JavaScript can change HTML content, styles, and attributes on the fly, making web pages more dynamic and interactive.
   2. This includes tasks like updating the content of a web page without reloading it, creating interactive forms, and implementing animations.
3. **Libraries and Frameworks:**
   1. JavaScript has a rich ecosystem of libraries (like jQuery) and frameworks (like React, Angular, and Vue.js) that simplify complex tasks and enhance development efficiency.
   2. These tools provide pre-built functions and components that speed up development.
4. **Server-Side Development:**
   1. With the introduction of Node.js, JavaScript can also be used for server-side development.
   2. This means developers can use JavaScript to write code that runs on servers, handling tasks like database interactions, server-side logic, and more.
5. **Asynchronous Programming:**
   1. JavaScript supports asynchronous programming through callbacks, promises, and async/await syntax.
   2. This allows developers to write non-blocking code that can handle multiple tasks simultaneously, improving performance and user experience.
6. **Versatility:**
   1. JavaScript is versatile and can be used for various tasks beyond web development, such as game development, mobile app development (using frameworks like React Native), and even desktop applications (using Electron).
7. **Event-Driven:**
   1. JavaScript is event-driven, meaning it can listen for and respond to events (e.g., user clicks, form submissions) in real-time, providing a responsive and interactive experience.
8. **Standardized:**
   1. JavaScript is standardized through the ECMAScript specification, which ensures consistency and compatibility across different browsers and platforms.

Overall, JavaScript is an essential language for modern web development, enabling developers to create rich, interactive experiences on the web.

**Contents of JavaScript**

1. **Introduction to JavaScript**
   * What is JavaScript?
   * Where to use JavaScript (Client-side, Server-side with Node.js)
   * JavaScript Execution Environment (Browser, Node.js)
   * Setting up Development Environment (Browser Developer Tools, Editors like VS Code)
2. **Data Types and Variables**
   * Primitive Data Types (String, Number, Boolean, Null, Undefined, Symbol, BigInt)
   * Complex Data Types (Objects, Arrays, Functions)
   * Variables (let, const, var)
   * Type Conversion and Coercion
3. **Operators**
   * Arithmetic, Comparison, Logical, Assignment, Bitwise Operators
   * Conditional (Ternary) Operator
   * Precedence and Associativity of Operators
4. **Control Structures**
   * Conditional Statements (if, else, switch)
   * Loops (for, while, do-while, for...of, for...in)
   * Iteration Protocols (Iterator, Iterable)
5. **Functions**
   * Function Declarations and Expressions
   * Function Parameters (default, rest parameters)
   * Return Statements and undefined
   * **Closures** (Lexical Environment, Access to Outer Scope)
   * Arrow Functions (=>)
   * Higher-order Functions (Functions that return or accept functions)
   * **Immediately Invoked Function Expressions (IIFE)**
6. **Objects and Arrays**
   * Object Literals and Constructors
   * Accessing, Adding, and Removing Properties
   * **Object Methods (this keyword)** and Method Binding
   * Arrays: Creating, Accessing, Modifying, Looping through Arrays
   * **Array Methods** (push, pop, map, filter, reduce, find, some, every, etc.)
   * Destructuring Assignment (Objects and Arrays)
   * **Spread Operator** and **Rest Operator**
   * **Shallow Copy vs. Deep Copy** of Objects and Arrays
7. **Prototype and Inheritance**
   * Prototypes and Prototype Chain
   * Inheritance in JavaScript
   * Object.create() and Prototype-based Inheritance
   * **Class Syntax** in ES6+
   * Constructors and super()
   * Inheritance in Classes
8. **DOM Manipulation**
   * Introduction to the DOM
   * Selecting Elements (getElementById, querySelector, querySelectorAll)
   * Modifying Element Content (innerHTML, textContent)
   * Modifying Element Attributes (setAttribute, removeAttribute)
   * Modifying Styles with JavaScript (style object)
   * Event Handling (addEventListener, Event Bubbling, Event Delegation)
   * Creating and Removing Elements (createElement, appendChild, removeChild)
9. **Forms and Input Validation**
   * Handling Forms in JavaScript
   * Retrieving Form Data
   * Validating Input (Basic Validation, Regular Expressions)
   * Preventing Default Form Submissions (preventDefault())
   * Creating Custom Validation Messages
10. **Object-Oriented Programming (OOP)**

* Constructors and Instances
* Class-Based vs. Prototype-Based Inheritance
* Encapsulation, Polymorphism, and Inheritance
* Getters and Setters
* Static Methods
* Composition vs. Inheritance

1. **Asynchronous JavaScript**

* **Callbacks**
* Promises (resolve, reject, then, catch, finally)
* async / await
* Error Handling in Asynchronous Code (try, catch)
* **Fetching Data from APIs** (using fetch and XMLHttpRequest)
* **AJAX** and HTTP Requests (GET, POST, PUT, DELETE)

1. **Error Handling**

* try, catch, finally Blocks
* Throwing Custom Errors
* **Creating Custom Error Types**

1. **JavaScript Modules**

* **ES6 Modules** (import/export)
* Default and Named Exports
* **CommonJS Modules** (used in Node.js)
* Bundling Modules (with tools like Webpack)

1. **Event Loop and JavaScript Engine**

* JavaScript's Single Threaded Nature
* Event Loop (Call Stack, Task Queue)
* **Microtasks** and **Macrotasks**
* Non-blocking I/O with Event Loop

1. **Local Storage, Session Storage, and Cookies**

* **localStorage** and **sessionStorage**
* Storing and Retrieving Data from Browser Storage
* JSON Storage in Local/Session Storage
* **Cookies** and Managing Cookies with JavaScript

1. **Regular Expressions**

* Regular Expressions Syntax (/pattern/flags)
* Commonly Used Methods (test, match, replace, search)
* Validating Patterns in Strings (e.g., emails, phone numbers)

1. **JSON and AJAX**

* Introduction to JSON (JavaScript Object Notation)
* Parsing JSON with JSON.parse()
* Stringifying JavaScript Objects with JSON.stringify()
* AJAX Requests and Interacting with REST APIs

1. **JavaScript Performance**

* Optimizing JavaScript Code (Execution, Memory, Reflows)
* **Debouncing** and **Throttling**
* Managing Heavy Computations in JavaScript (e.g., Web Workers)

1. **JavaScript Design Patterns**

* Singleton Pattern
* Factory Pattern
* Observer Pattern
* Module Pattern
* Revealing Module Pattern

1. **Testing JavaScript**

* **Unit Testing** (Jest, Mocha, Jasmine)
* Writing Testable Code
* Asynchronous Testing

1. **Advanced Topics**

* **Proxies** and Reflect API
* Generators and yield
* Iterators and Iterables
* Symbol and its use cases
* Web Workers for Multithreading
* **WeakMap** and **WeakSet**

1. **JavaScript in Front-End Frameworks**

* Introduction to Modern JavaScript Frameworks (React, Angular, Vue.js)
* Setting up Frameworks with JavaScript
* Component-Based Architecture in Frameworks

**Basics of Javascript**

**1. Syntax and Variables**

**Variables** are used to store data values.

You can declare a variable using **var**, **let**, or **const**.

var name = "John"; // Declares a variable that can be reassigned

let age = 25; // Declares a block-scoped variable that can be reassigned

const pi = 3.14; // Declares a block-scoped variable that cannot be reassigned

**2. Data Types**

JavaScript supports various data types, including:

* **Primitive Types:**
  + String: "Hello"
  + Number: 42
  + Boolean: true or false
  + Null: null
  + Undefined: undefined
  + Symbol: Symbol('id')
  + BigInt: 123n
* **Composite Types:**
  + Object: { name: "John", age: 25 }
  + Array: [1, 2, 3]

**3. Operators**

JavaScript provides various operators:

* **Arithmetic Operators:** +, -, \*, /, %
* **Assignment Operators:** =, +=, -=, \*=, /=
* **Comparison Operators:** ==, ===, !=, !==, >, <, >=, <=
* **Logical Operators:** &&, ||, !

**4. Control Structures(decisions)**

**Conditional Statements** allow you to execute code based on conditions.

if (age > 18) {

console.log("Adult");

} else {

console.log("Minor");

}

**Loops** allow you to execute code multiple times.

* **For Loop:**

for (let i = 0; i < 5; i++) {

console.log(i);

}

* **While Loop:**

let i = 0;

while (i < 5) {

console.log(i);

i++;

}

**5. Functions**

Functions are reusable blocks of code.

function greet(name) {

return "Hello, " + name;

}

console.log(greet("Alice"));

**6. Objects and Arrays**

**Objects** store key-value pairs.

let person = {

name: "John",

age: 25,

greet: function() {

console.log("Hello, " + this.name);

}

};

person.greet();

**Arrays** store ordered collections of values.

let numbers = [1, 2, 3, 4, 5];

console.log(numbers[0]); // Outputs 1

numbers.push(6); // Adds a new element to the array

console.log(numbers);

**7. Event Handling**

JavaScript can handle events like user actions.

document.getElementById("myButton").addEventListener("click", function() {

alert("Button was clicked!");

});

**8. DOM Manipulation**

JavaScript can manipulate the Document Object Model (DOM) to change the content of a webpage.

document.getElementById("myElement").innerHTML = "New Content";

document.getElementById("myElement").style.color = "red";

**9. Promises and Async/Await**

JavaScript supports asynchronous programming using Promises and the async/await syntax.

* **Promise:**

let promise = new Promise(function(resolve, reject) {

// Some async operation

if (success) {

resolve("Success");

} else {

reject("Error");

}

});

promise.then(function(result) {

console.log(result);

}).catch(function(error) {

console.error(error);

});

* **Async/Await:**

async function fetchData() {

try {

let response = await fetch('https://api.example.com/data');

let data = await response.json();

console.log(data);

} catch (error) {

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

}

}

fetchData();

These basics provide a foundation for learning JavaScript. As you practice and build more projects, you'll become more comfortable with its syntax and capabilities.

**document.querySelector()**

The document.querySelector() method in JavaScript is used to select the first element in the DOM that matches a specified CSS selector.

It allows you to find and manipulate specific HTML elements based on their tag name, class, ID, or other attributes.

**Syntax:**

let element = document.querySelector(selector);

**Parameters:**

* **selector**: A string containing one or more CSS selectors to match the desired element. It can be:
  + **Tag** **name**: "div", "h1", etc.
  + **Class** **name**: ".class-name"
  + **ID**: "#id-name"
  + **Attribute**: "[attribute=value]"

**Example Usage:**

1. **Select an element by class**:

let element = document.querySelector('.myClass');

1. **Select an element by ID**:

let element = document.querySelector('#myId');

1. **Select a nested element**:

let element = document.querySelector('div > p');

1. **Select an input with a specific attribute**:

let element = document.querySelector('input[type="text"]');

**Key Points:**

* **First Match**: It returns only the first element that matches the selector. If no match is found, it returns null.
* **CSS Selectors**: It supports complex CSS selectors, so you can target elements more precisely.

**Example:**

<!DOCTYPE html>

<html>

<head>

<title>querySelector Example</title>

</head>

<body>

<div class="container">

<p id="first-paragraph">This is the first paragraph.</p>

<p>This is the second paragraph.</p>

</div>

<script>

let firstParagraph = document.querySelector('#first-paragraph');

console.log(firstParagraph.textContent); // Outputs: "This is the first paragraph."

</script>

</body>

</html>

In this example, document.querySelector('#first-paragraph') selects the paragraph with the id="first-paragraph" and logs its content.

**Decisions in Javascript**

In JavaScript, decision-making is primarily handled through conditional statements.

These statements allow you to execute different blocks of code based on certain conditions.

The main types of conditional statements in JavaScript are if, else if, else, and switch.

Here’s a detailed look at each:

**1. if Statement**

The if statement is used to execute a block of code if a specified condition is true.

let age = 20;

if (age >= 18) {

console.log("You are an adult.");

}

**2. else Statement**

The else statement is used to execute a block of code if the condition in the if statement is false.

let age = 15;

if (age >= 18) {

console.log("You are an adult.");

} else {

console.log("You are a minor.");

}

**3. else if Statement**

The else if statement is used to specify a new condition if the first condition is false.

let score = 85;

if (score >= 90) {

console.log("Grade: A");

} else if (score >= 80) {

console.log("Grade: B");

} else if (score >= 70) {

console.log("Grade: C");

} else {

console.log("Grade: F");

}

**4. switch Statement**

The switch statement is used to perform different actions based on different conditions. It is a good alternative when you need to compare the same value to several different possible matches.

let day = 3;

let dayName;

switch (day) {

case 1:

dayName = "Monday";

break;

case 2:

dayName = "Tuesday";

break;

case 3:

dayName = "Wednesday";

break;

case 4:

dayName = "Thursday";

break;

case 5:

dayName = "Friday";

break;

case 6:

dayName = "Saturday";

break;

case 7:

dayName = "Sunday";

break;

default:

dayName = "Invalid day";

}

console.log(dayName); // Outputs: Wednesday

**5. Ternary Operator**

The ternary operator is a shorthand for the if-else statement. It is useful for making simple decisions.

let age = 20;

let access = (age >= 18) ? "Granted" : "Denied";

console.log(access); // Outputs: Granted

**Example: Combining Conditions**

You can combine multiple conditions using logical operators like && (AND), || (OR), and ! (NOT).

let age = 25;

let hasPermission = true;

if (age >= 18 && hasPermission) {

console.log("Access granted.");

} else {

console.log("Access denied.");

}

if (age < 18 || !hasPermission) {

console.log("Access denied.");

} else {

console.log("Access granted.");

}

**Summary**

* Use if to specify a block of code to be executed if a condition is true.
* Use else to specify a block of code to be executed if the same condition is false.
* Use else if to specify a new condition to test if the first condition is false.
* Use switch to specify many alternative blocks of code to be executed.
* Use the ternary operator for simple conditional expressions.

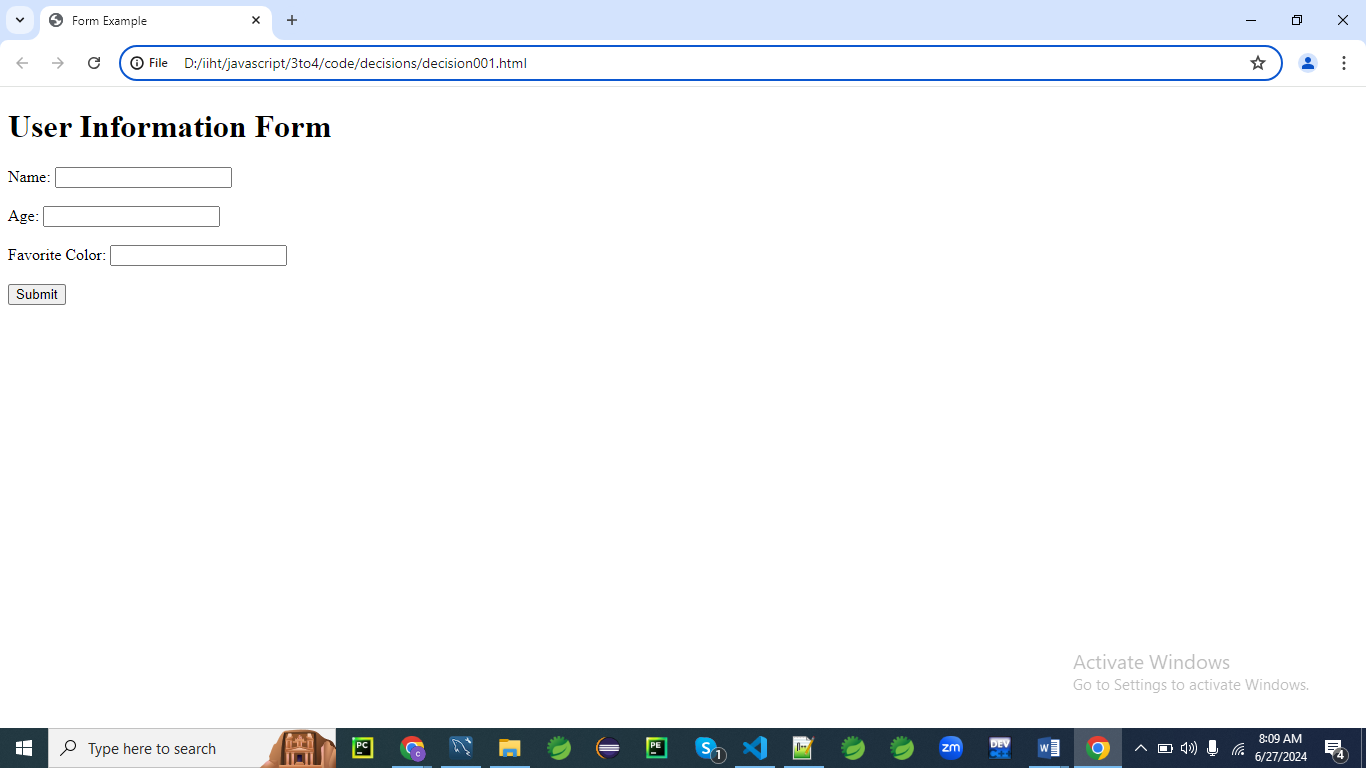
Understanding these basic decision-making structures will allow you to write more dynamic and responsive JavaScript code.

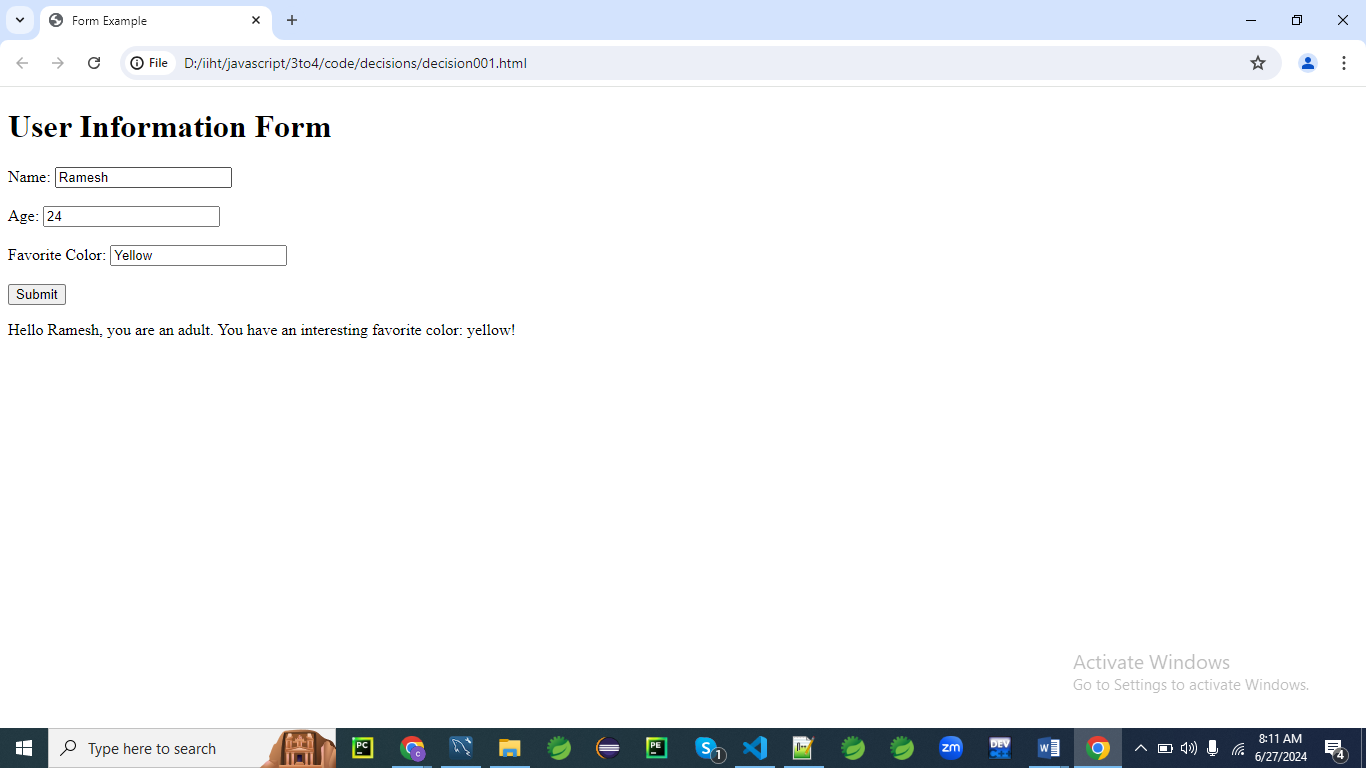
**Example 01**

Here's an example program in JavaScript that uses a form to demonstrate decision-making.

This program will validate user input from a form, checking the age and favorite color provided by the user.

Depending on the input, it will display different messages.





**HTML Form**

First, let's create an HTML form with fields for name, age, and favorite color.

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

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

<title>Form Example</title>

</head>

<body>

<h1>User Information Form</h1>

<form id="userForm">

<label for="name">Name:</label>

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

<label for="age">Age:</label>

<input type="number" id="age" name="age" required><br><br>

<label for="color">Favorite Color:</label>

<input type="text" id="color" name="color" required><br><br>

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

</form>

<p id="message"></p>

<script src="script001.js"></script>

</body>

</html>

**JavaScript Code(script001.js)**

Now, let's write the JavaScript code that handles the form submission and includes various decision-making structures.

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

event.preventDefault(); // Prevent the form from submitting the traditional way

// Get form values

let name = document.getElementById('name').value;

let age = parseInt(document.getElementById('age').value);

let color = document.getElementById('color').value.toLowerCase();

// Decision-making logic

let message = '';

if (age >= 0 && age < 18) {

message = `Hello ${name}, you are a minor.`;

}

else if (age >= 18 && age < 60) {

message = `Hello ${name}, you are an adult.`;

}

else if (age >= 60) {

message = `Hello ${name}, you are a senior citizen.`;

}

else {

message = `Invalid age entered.`;

}

// Using switch-case for favorite color

switch (color) {

case 'red':

message += ' Your favorite color is Red!';

break;

case 'blue':

message += ' Your favorite color is Blue!';

break;

case 'green':

message += ' Your favorite color is Green!';

break;

default:

message += ` You have an interesting favorite color: ${color}!`;

}

// Display the message

document.getElementById('message').innerText = message;

});

**Explanation**

1. **HTML Form:**
   * The form contains three input fields: name, age, and favorite color.
   * When the form is submitted, the submit event is triggered.
2. **JavaScript Code:**
   * An event listener is added to handle the form submission.
   * The event.preventDefault() method prevents the form from submitting in the traditional way, allowing us to handle the submission with JavaScript.
   * The values from the form fields are retrieved and stored in variables.
   * Conditional statements (if, else if, else) are used to determine the message based on the user's age.
   * A switch statement is used to add a specific message based on the user's favorite color.
   * The final message is displayed in the paragraph with the id message.

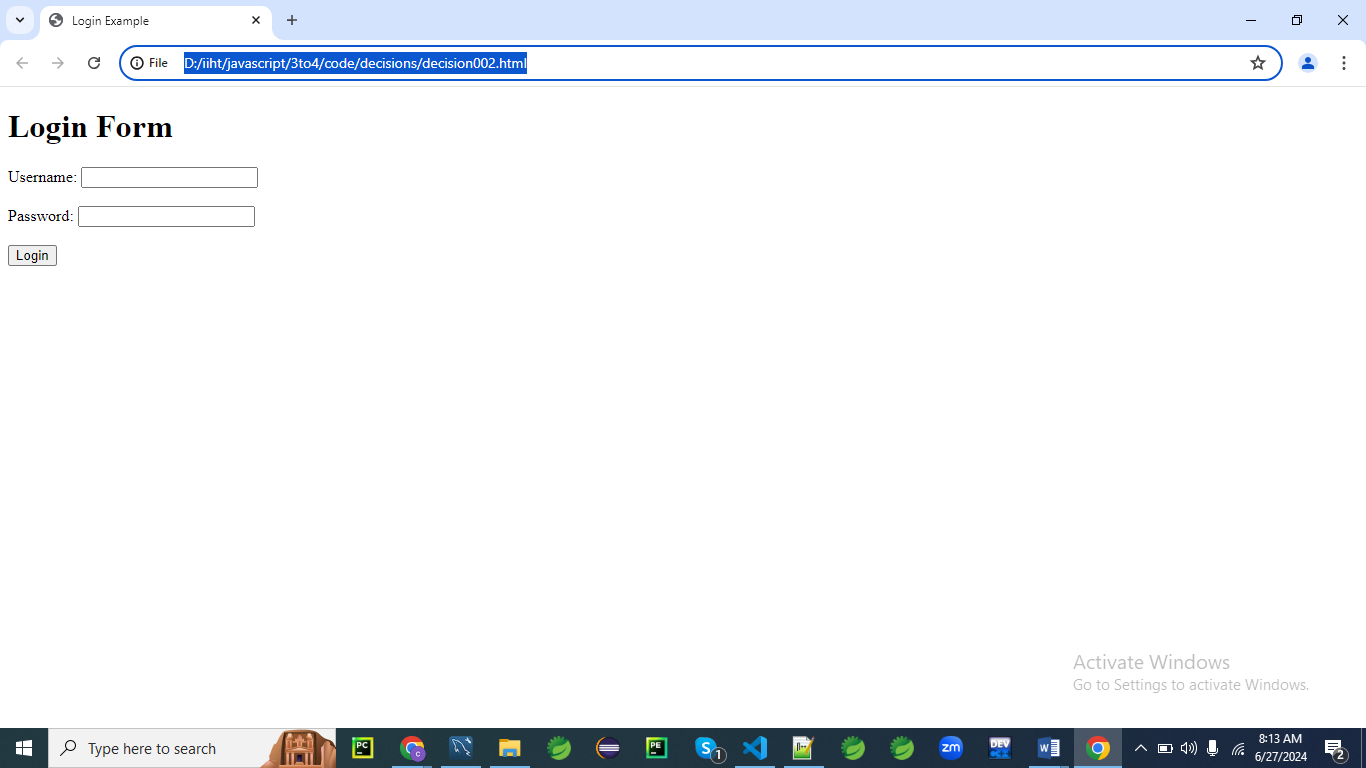
This example demonstrates how to use various decision-making structures in JavaScript to handle and validate form input.

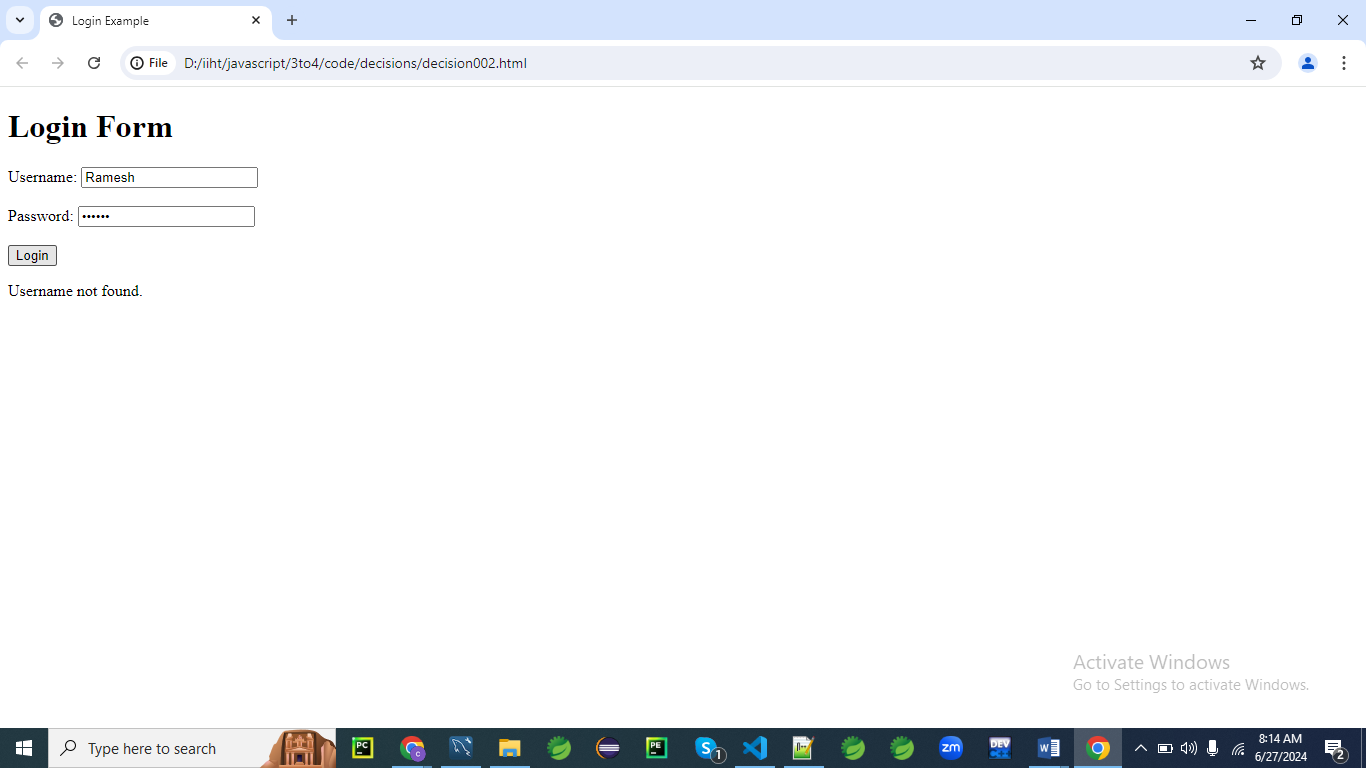
**Example 02**

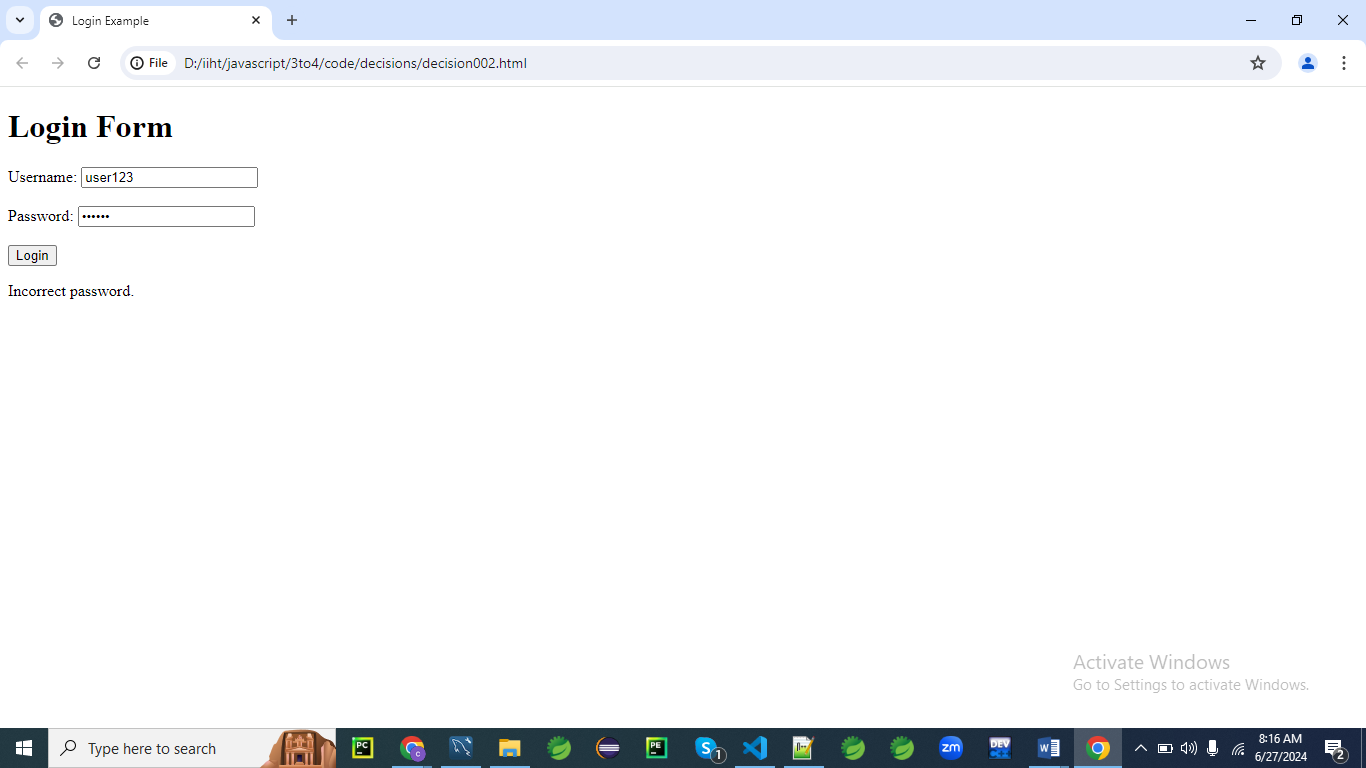
Here's another example program in JavaScript that uses a form to demonstrate decision-making.

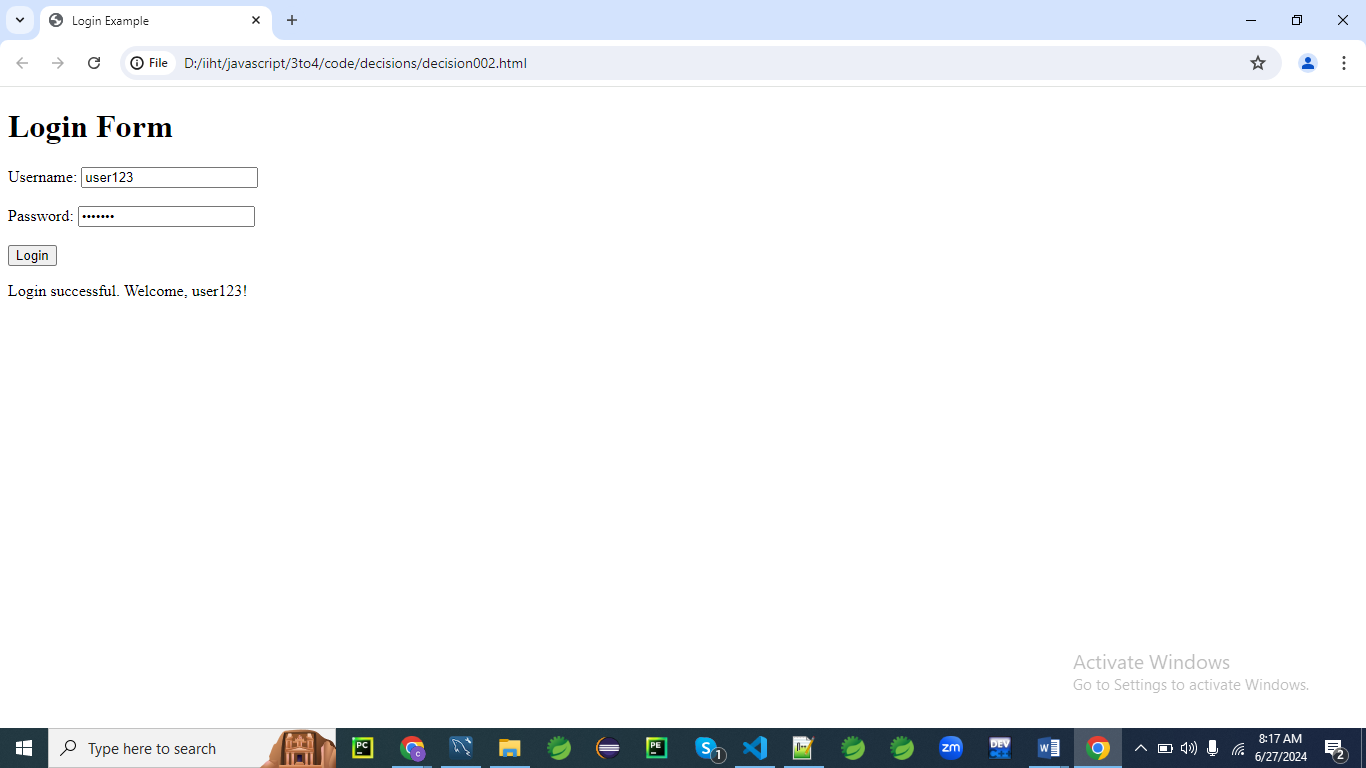
This example will include a login form where the user must enter a username and password.

The program will check the input against predefined values and display appropriate messages based on the validity of the credentials.









**HTML Form**

First, let's create an HTML form with fields for username and password.

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

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

<title>Login Example</title>

</head>

<body>

<h1>Login Form</h1>

<form id="loginForm">

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

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

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

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

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

</form>

<p id="message"></p>

<script src="script002.js"></script>

</body>

</html>

**JavaScript Code**

Now, let's write the JavaScript code that handles the form submission and includes various decision-making structures.

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

event.preventDefault(); // Prevent the form from submitting the traditional way

// Get form values

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

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

// Predefined valid credentials

let validUsername = "user123";

let validPassword = "pass123";

// Decision-making logic

let message = '';

// Check if the username is correct

if (username === validUsername) {

// Check if the password is correct

if (password === validPassword) {

message = "Login successful. Welcome, " + username + "!";

} else {

message = "Incorrect password.";

}

} else {

message = "Username not found.";

}

// Display the message

document.getElementById('message').innerText = message;

});

**Explanation**

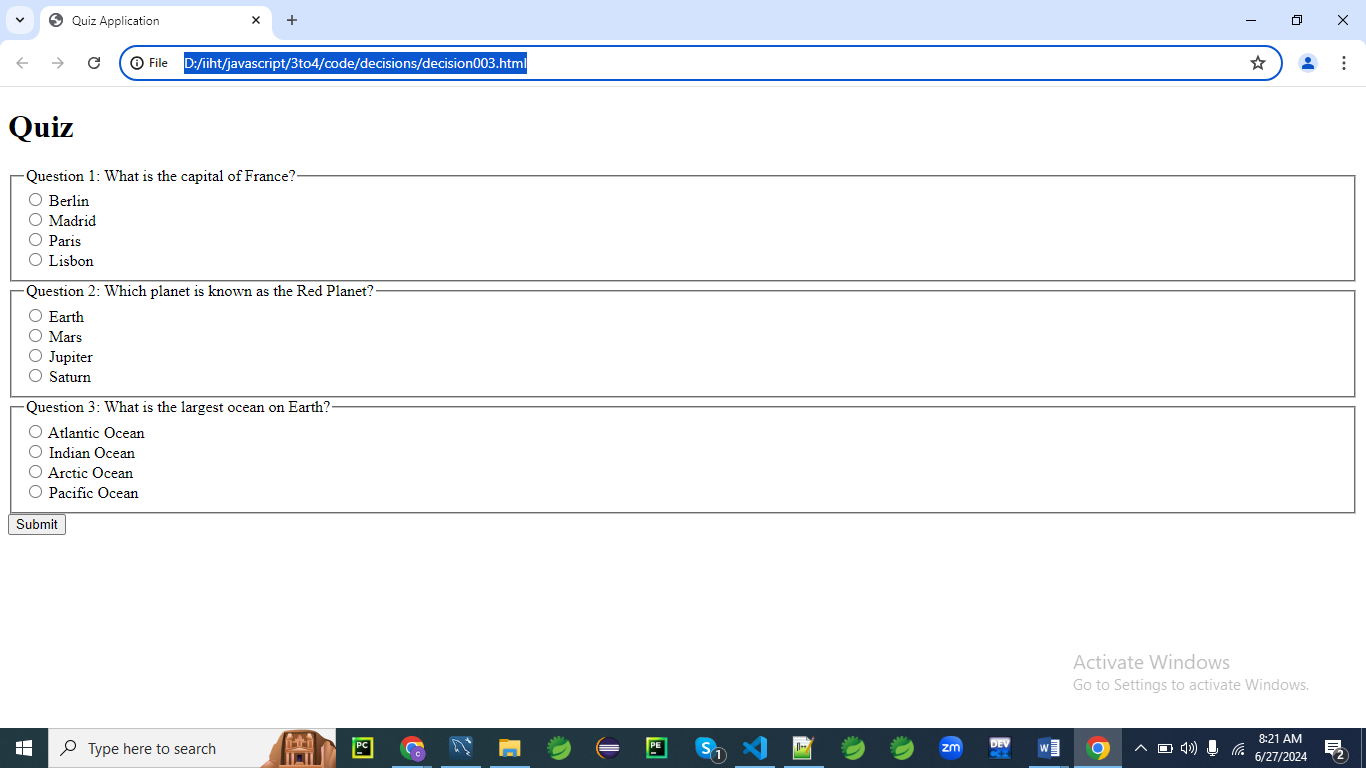
1. **HTML Form:**
   * The form contains two input fields: username and password.
   * When the form is submitted, the submit event is triggered.
2. **JavaScript Code:**
   * An event listener is added to handle the form submission.
   * The event.preventDefault() method prevents the form from submitting in the traditional way, allowing us to handle the submission with JavaScript.
   * The values from the form fields are retrieved and stored in variables.
   * Predefined valid credentials are defined for comparison.
   * Conditional statements (if, else) are used to validate the username and password.
     + First, the username is checked. If it matches the predefined valid username, the password is checked next.
     + If both the username and password are correct, a success message is displayed.
     + If the username is correct but the password is incorrect, an appropriate error message is displayed.
     + If the username is incorrect, a different error message is displayed.
   * The final message is displayed in the paragraph with the id message.

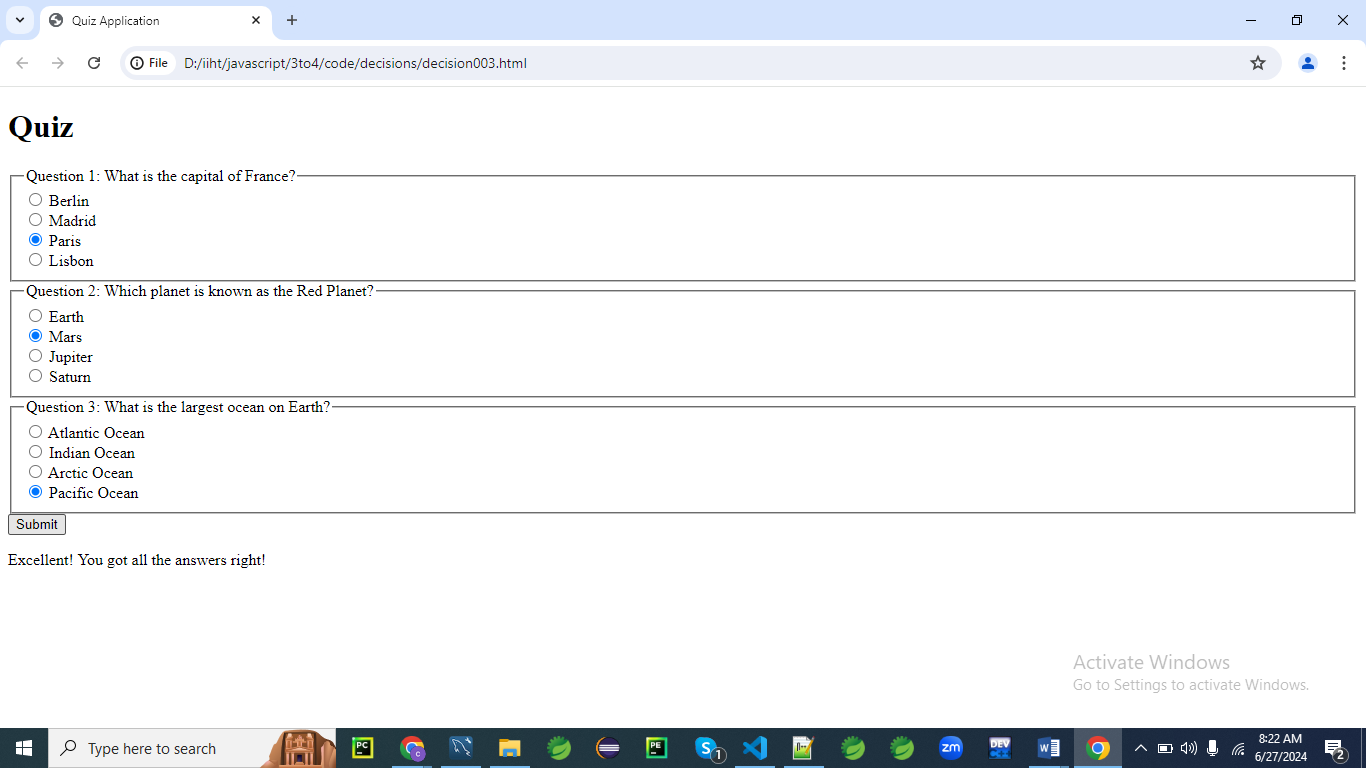
This example demonstrates how to use nested decision-making structures in JavaScript to handle and validate login credentials

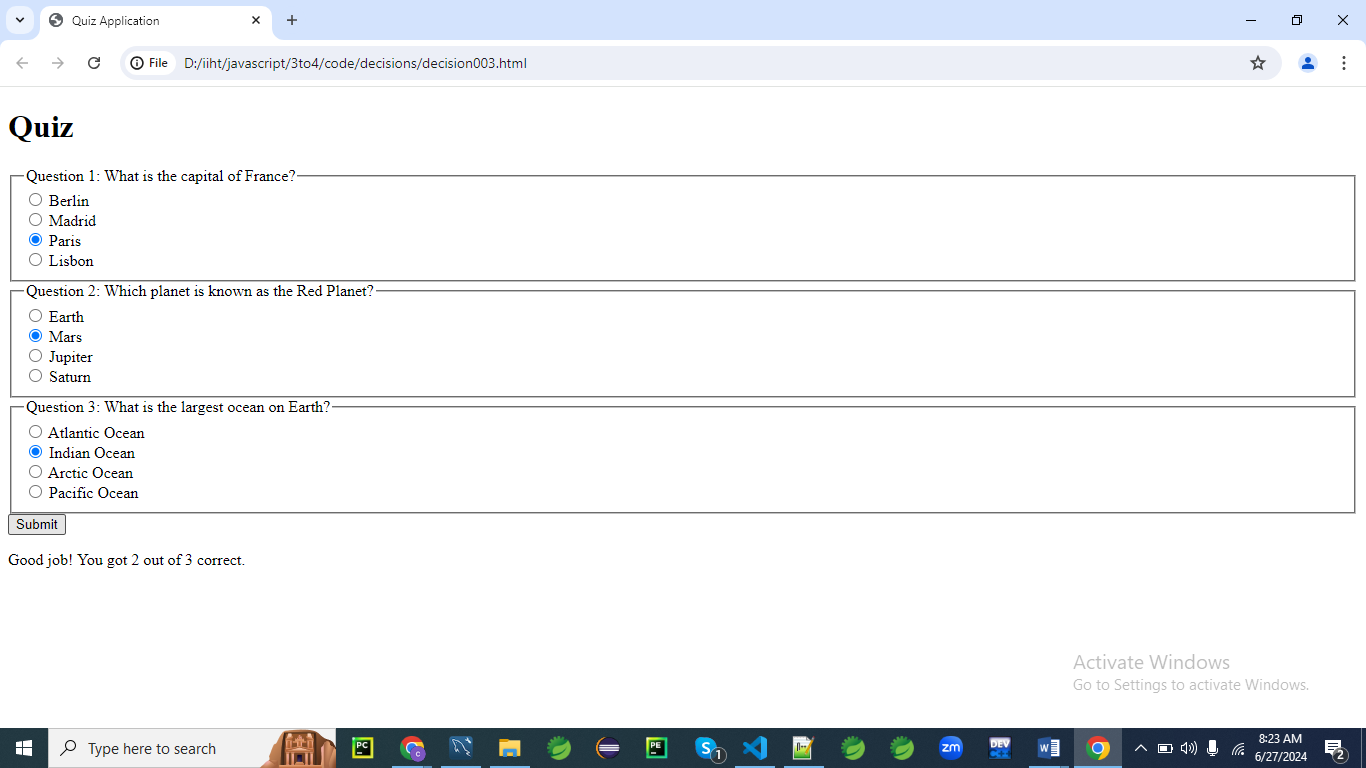
**Example 03**

Let's create a more comprehensive example program in JavaScript that demonstrates a variety of decision-making structures.

This example will be a simple quiz application where users can answer multiple-choice questions, and the application will provide feedback based on their answers.







**HTML Form**

First, let's create an HTML form for the quiz with multiple-choice questions.

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

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

<title>Quiz Application</title>

</head>

<body>

<h1>Quiz</h1>

<form id="quizForm">

<fieldset>

<legend>Question 1: What is the capital of France?</legend>

<label>

<input type="radio" name="q1" value="Berlin" required> Berlin

</label><br>

<label>

<input type="radio" name="q1" value="Madrid"> Madrid

</label><br>

<label>

<input type="radio" name="q1" value="Paris"> Paris

</label><br>

<label>

<input type="radio" name="q1" value="Lisbon"> Lisbon

</label><br>

</fieldset>

<fieldset>

<legend>Question 2: Which planet is known as the Red Planet?</legend>

<label>

<input type="radio" name="q2" value="Earth" required> Earth

</label><br>

<label>

<input type="radio" name="q2" value="Mars"> Mars

</label><br>

<label>

<input type="radio" name="q2" value="Jupiter"> Jupiter

</label><br>

<label>

<input type="radio" name="q2" value="Saturn"> Saturn

</label><br>

</fieldset>

<fieldset>

<legend>Question 3: What is the largest ocean on Earth?</legend>

<label>

<input type="radio" name="q3" value="Atlantic" required> Atlantic Ocean

</label><br>

<label>

<input type="radio" name="q3" value="Indian"> Indian Ocean

</label><br>

<label>

<input type="radio" name="q3" value="Arctic"> Arctic Ocean

</label><br>

<label>

<input type="radio" name="q3" value="Pacific"> Pacific Ocean

</label><br>

</fieldset>

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

</form>

<p id="message"></p>

<script src="script.js"></script>

</body>

</html>

**JavaScript Code**

Now, let's write the JavaScript code that handles the form submission and includes various decision-making structures.

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

event.preventDefault(); // Prevent the form from submitting the traditional way

// Get form values

let q1 = document.querySelector('input[name="q1"]:checked').value;

let q2 = document.querySelector('input[name="q2"]:checked').value;

let q3 = document.querySelector('input[name="q3"]:checked').value;

// Correct answers

// Create an object correctAnswers

let correctAnswers = {

q1: "Paris",

q2: "Mars",

q3: "Pacific"

};

// Decision-making logic

let score = 0;

if (q1 === correctAnswers.q1) {

score++;

}

if (q2 === correctAnswers.q2) {

score++;

}

if (q3 === correctAnswers.q3) {

score++;

}

// Using switch-case for feedback based on score

let feedback = '';

switch (score) {

case 3:

feedback = "Excellent! You got all the answers right!";

break;

case 2:

feedback = "Good job! You got 2 out of 3 correct.";

break;

case 1:

feedback = "You got 1 out of 3 correct. Keep trying!";

break;

case 0:

feedback = "You got all the answers wrong. Better luck next time!";

break;

default:

feedback = "An unexpected error occurred.";

}

// Display the feedback

document.getElementById('message').innerText = feedback;

});

**Explanation**

1. **HTML Form:**
   * The form contains three multiple-choice questions, each with four possible answers.
   * Each question uses radio buttons to allow the user to select one answer.
   * When the form is submitted, the submit event is triggered.
2. **JavaScript Code:**
   * An event listener is added to handle the form submission.
   * The event.preventDefault() method prevents the form from submitting in the traditional way, allowing us to handle the submission with JavaScript.
   * The values of the selected radio buttons are retrieved and stored in variables.
   * Correct answers are predefined in an object.
   * Conditional statements (if) are used to compare the user's answers with the correct answers and calculate the score.
   * A switch statement is used to provide feedback based on the score.
   * The feedback message is displayed in the paragraph with the id message.

This example demonstrates how to use various decision-making structures in JavaScript to handle and validate user input in a quiz application. It includes if statements for condition checking and a switch statement for providing different feedback messages based on the user's score.

**Loops in javascript**

Loops are a fundamental part of JavaScript and allow you to execute a block of code multiple times.

JavaScript supports several types of loops: for, while, do...while, and for...in/for...of.

Here’s an overview of each type of loop along with examples.

**1. for Loop**

The for loop is used to execute a block of code a specific number of times.

It consists of three parts: initialization, condition, and iteration.

for (let i = 0; i < 5; i++) {

console.log("Iteration number: " + i);

}

**2. while Loop**

The while loop executes a block of code as long as a specified condition is true.

let i = 0;

while (i < 5) {

console.log("Iteration number: " + i);

i++;

}

**3. do...while Loop**

The do...while loop is similar to the while loop, but it executes the block of code at least once before checking the condition.

let i = 0;

do {

console.log("Iteration number: " + i);

i++;

} while (i < 5);

**4. for...in Loop**

The for...in loop is used to iterate over the properties of an object.

const person = { name: "Alice", age: 25, city: "New York" };

for (let key in person) {

console.log(key + ": " + person[key]);

}

**5. for...of Loop**

The for...of loop is used to iterate over iterable objects (like arrays, strings, etc.).

const numbers = [1, 2, 3, 4, 5];

for (let number of numbers) {

console.log("Number: " + number);

}

**6. Nested Loops**

Loops can also be nested, meaning you can have a loop inside another loop.

for (let i = 0; i < 3; i++) {

console.log("Outer loop iteration: " + i);

for (let j = 0; j < 2; j++) {

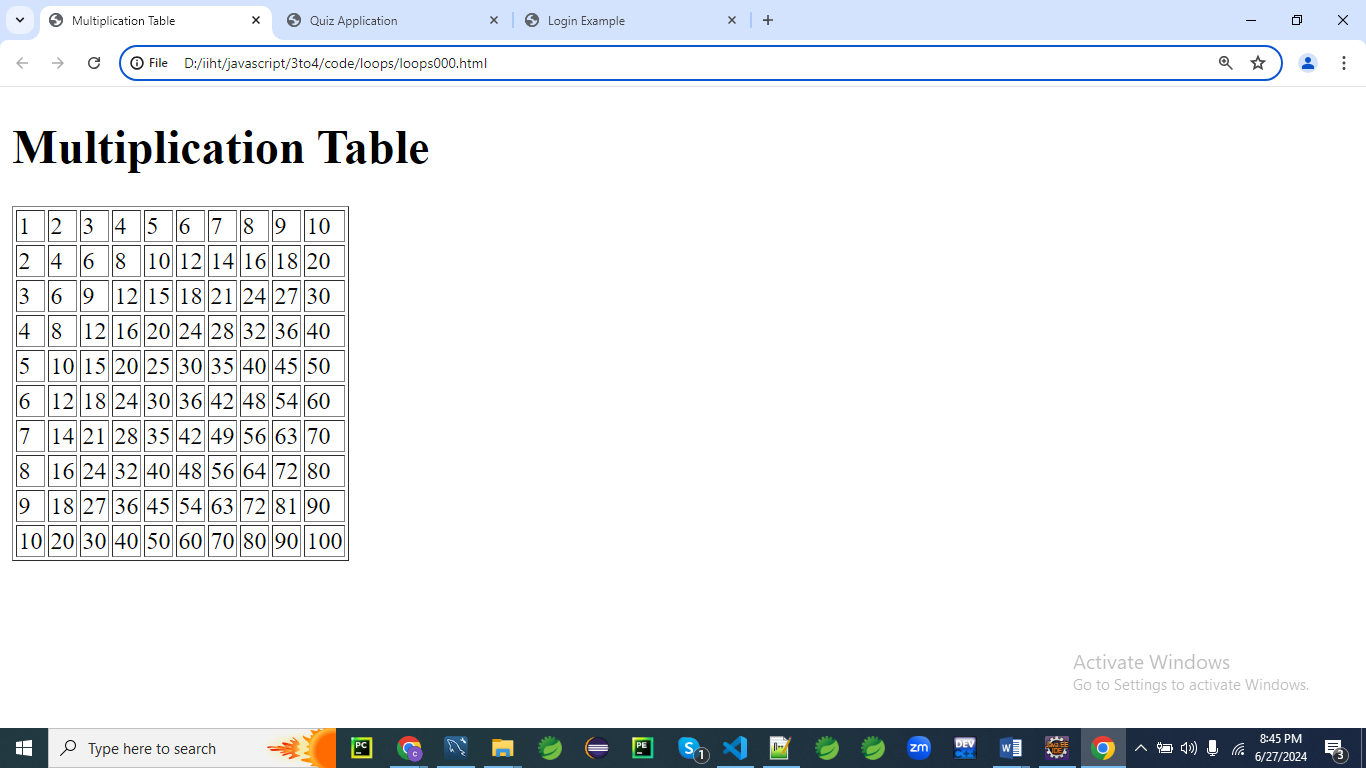
console.log(" Inner loop iteration: " + j);

}

}

**Example Program: Generating a Multiplication Table**

Here’s a complete example program that uses nested loops to generate a multiplication table.



<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

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

<title>Multiplication Table</title>

</head>

<body>

<h1>Multiplication Table</h1>

<table border="1" id="multiplicationTable"></table>

<script>

const table = document.getElementById('multiplicationTable');

for (let i = 1; i <= 10; i++) {

const row = table.insertRow();

for (let j = 1; j <= 10; j++) {

const cell = row.insertCell();

cell.innerText = i \* j;

}

}

</script>

</body>

</html>

**Explanation**

1. **HTML Structure:**
   * The HTML includes a table element where the multiplication table will be generated.
2. **JavaScript Code:**
   * The outer for loop iterates from 1 to 10 to create the rows of the table.
   * The inner for loop also iterates from 1 to 10 to create the cells of each row.
   * Each cell is filled with the product of the current row and column indices (i \* j).

This example demonstrates how to use nested loops to create a structured output, such as a multiplication table, dynamically in the HTML document.

**What do you mean DOMContentLoaded?**

The **DOMContentLoaded** event in JavaScript is fired when the **initial** HTML document has been completely loaded and parsed, without waiting for stylesheets, images, and subframes to finish loading.

This means the event is triggered as soon as the HTML has been fully loaded and the DOM (Document Object Model) has been constructed, allowing scripts to safely interact with the DOM.

Here's a more detailed explanation:

1. **Loading Process of a Web Page:**
   * **HTML Parsing:** The browser parses the HTML document and constructs the DOM.
   * **External Resources:** During the parsing of the HTML, if the browser encounters <script>, <link>, <img>, etc., it requests these resources.
   * **DOMContentLoaded Event:** This event fires when the HTML is completely parsed and the DOM is fully constructed, but before all external resources like images and stylesheets are loaded.
   * **load Event:** This event fires after the DOMContentLoaded event, once the entire page including all dependent resources (stylesheets, images) has fully loaded.
2. **Usage:**
   * The **DOMContentLoaded** event is useful when you want to execute JavaScript code as soon as the DOM is ready for manipulation, without waiting for the entire page to load.
   * It's commonly used to initialize scripts that rely on the DOM elements being present.

Here is a simple example to illustrate its use:

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<title>DOMContentLoaded Example</title>

</head>

<body>

<div id="content">Hello, World!</div>

<script>

document.addEventListener('DOMContentLoaded', function() {

// This code will run when the DOM is fully loaded

const contentDiv = document.getElementById('content');

contentDiv.textContent = 'DOM fully loaded and parsed!';

});

</script>

</body>

</html>

In this example:

* The DOMContentLoaded event listener is set up to run a function once the DOM is fully loaded.
* When the DOM is ready, it changes the text content of the <div id="content"> element.

**Why Use DOMContentLoaded?**

1. **Safe DOM Manipulation:** Ensures that your JavaScript code that manipulates the DOM will only run after the DOM is ready.
2. **Performance:** It allows your script to run sooner than waiting for the load event, which can improve the perceived performance of your web application.
3. **Avoiding Errors:** Prevents errors that occur when trying to access DOM elements that aren't yet available.

**Summary**

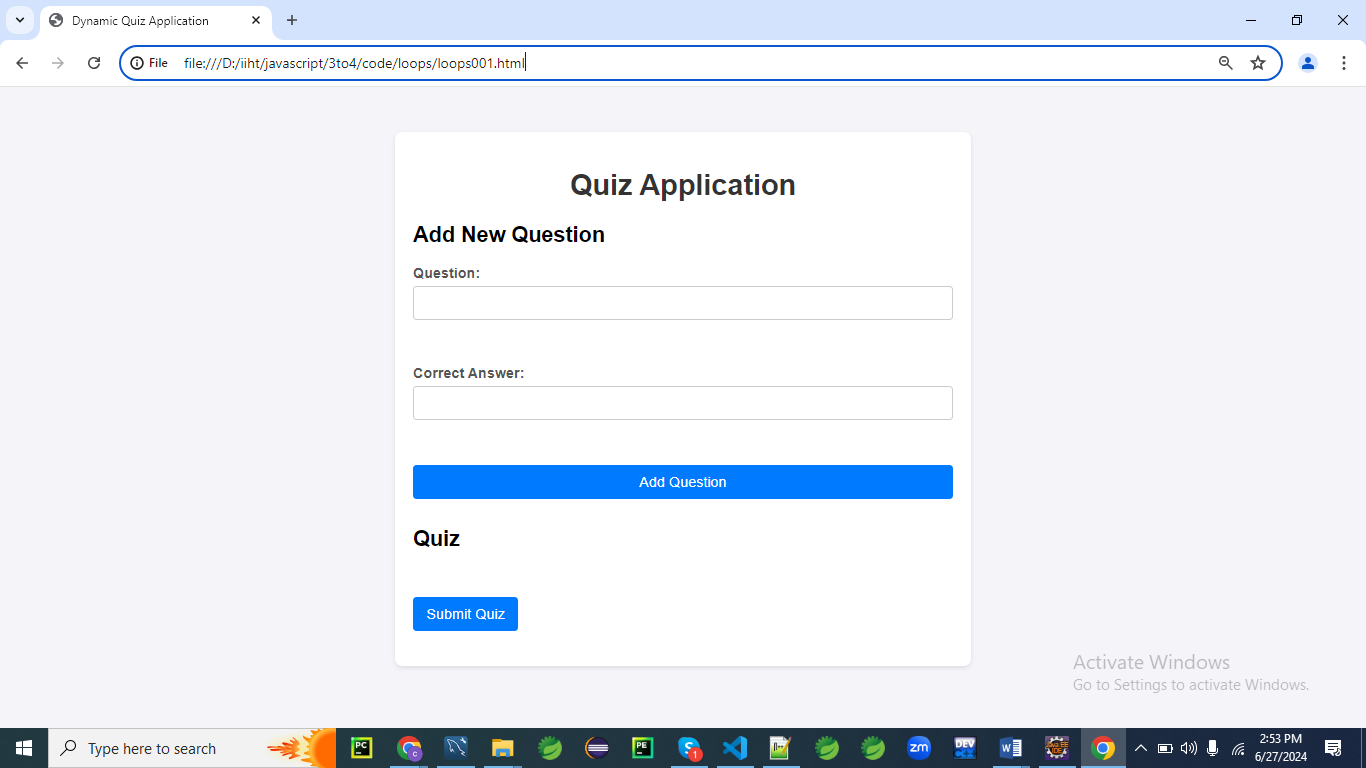
The DOMContentLoaded event is essential for writing scripts that interact with the DOM as soon as it's ready, ensuring that the page's structure is fully available for manipulation, without waiting for all the assets like images to load

**Example 01:**

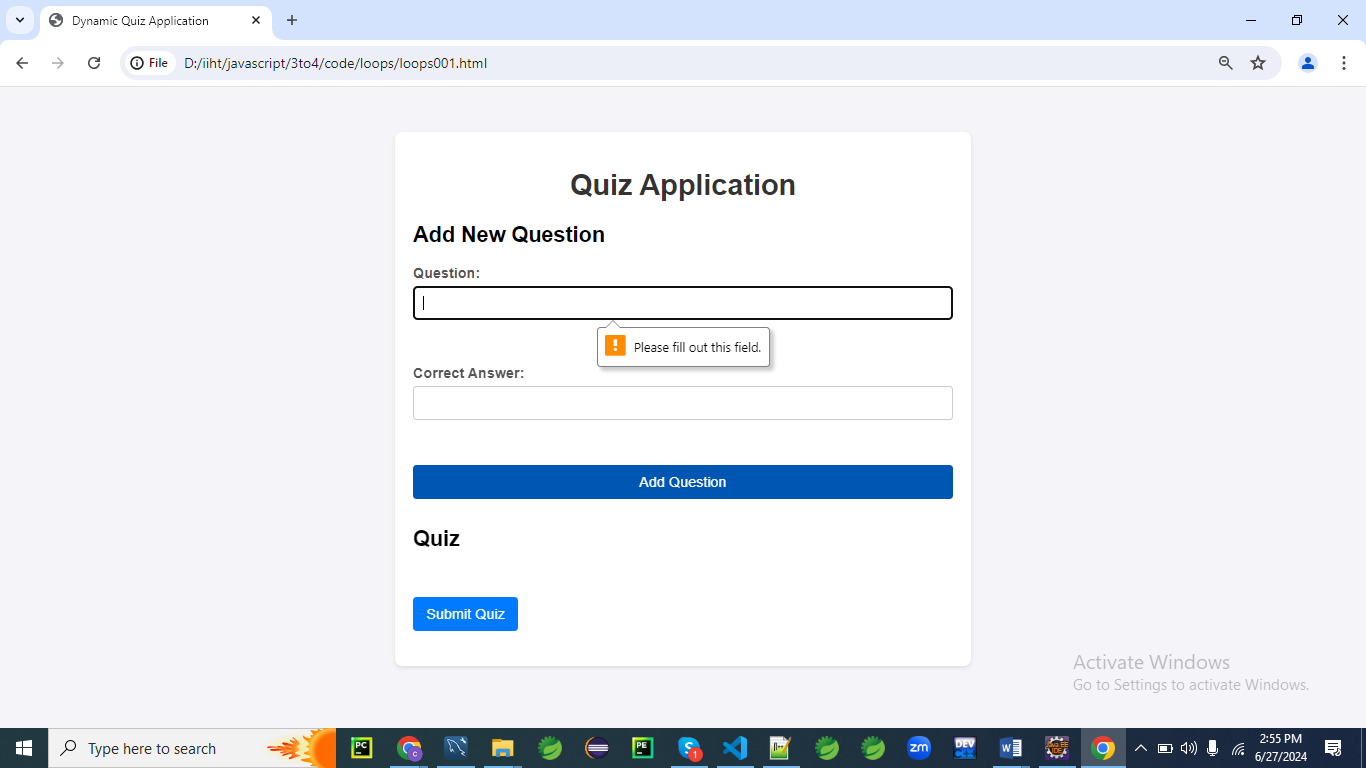
Let's create a comprehensive example that involves forms and loops with enhanced CSS3 styling.

This example will be a dynamic quiz application where users can add questions, take the quiz, and receive feedback based on their answers.

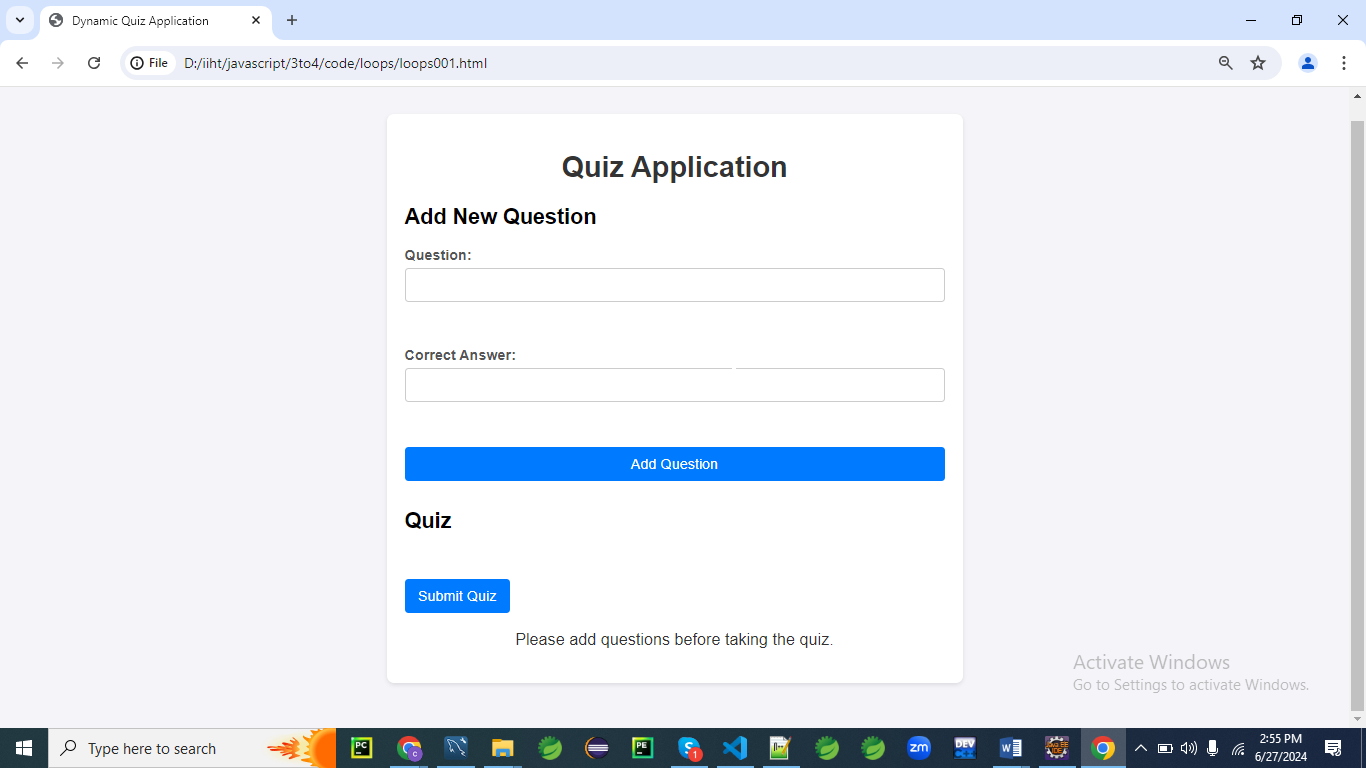
The application will use loops for rendering the questions, handling form submissions, and calculating results, and will also feature a modern, visually appealing design using CSS3.



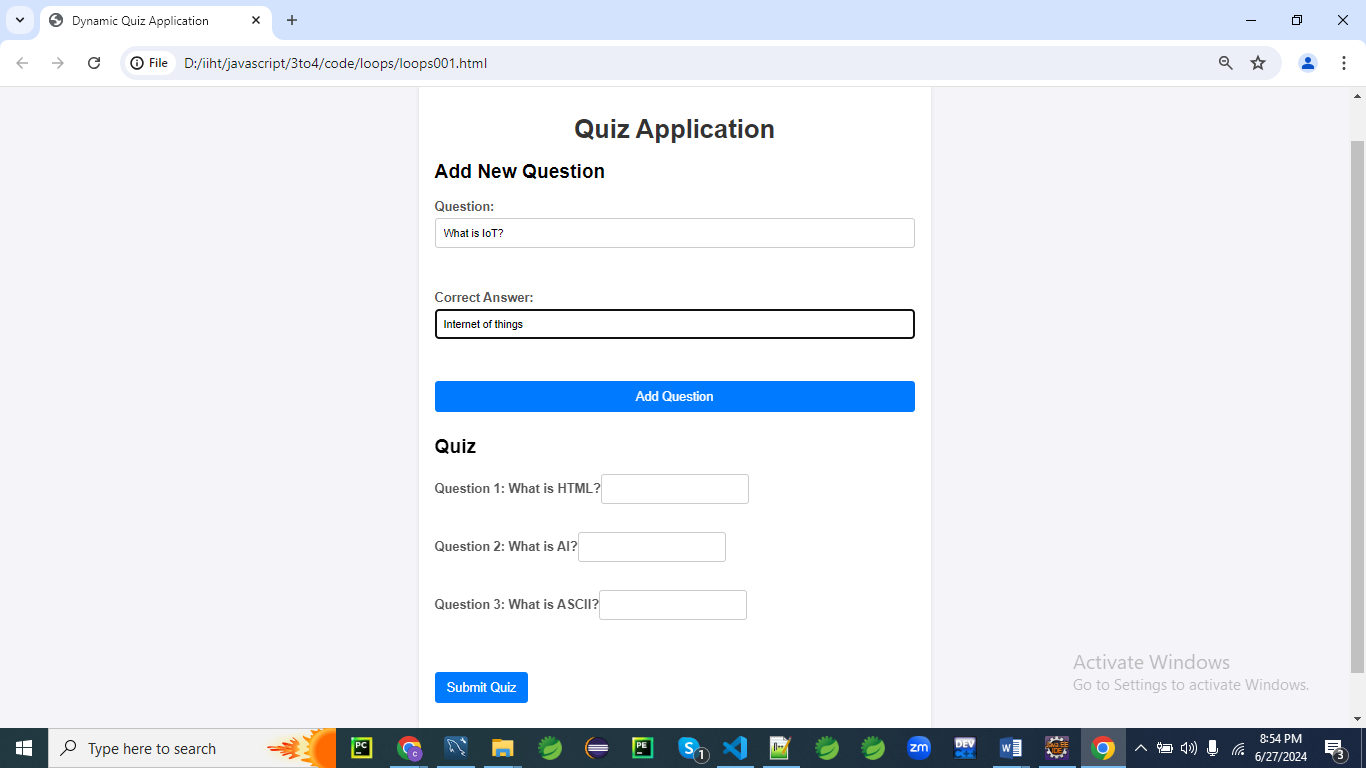
**When you click Add Question button**



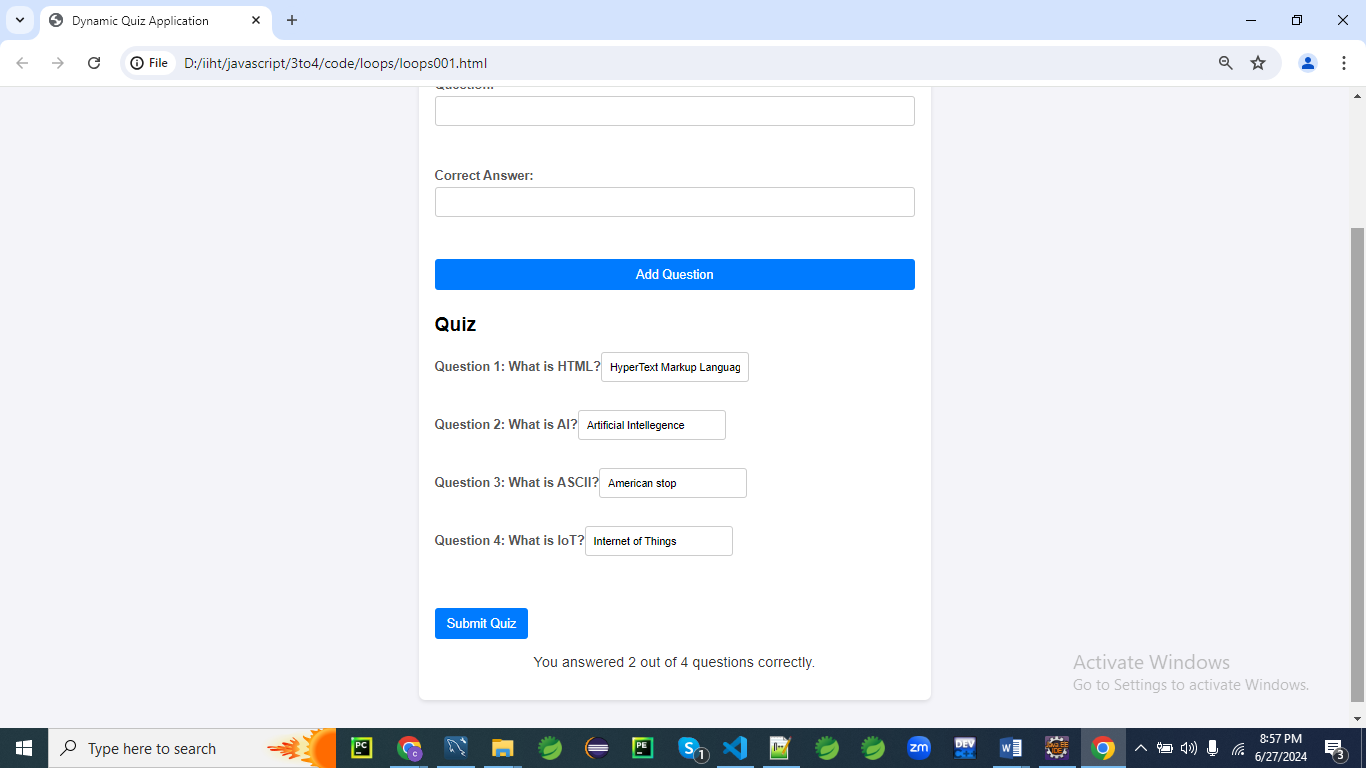
**When you click Submit Quiz button without filling the form**



**When you give the data and submit.**



**Final output**



**HTML Form**

First, let's create an HTML structure for the quiz application.

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

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

<title>Dynamic Quiz Application</title>

<link rel="stylesheet" href="styles003.css">

</head>

<body>

<div class="container">

<h1>Quiz Application</h1>

<!-- Form to add new questions -->

<div class="form-section">

<h2>Add New Question</h2>

<form id="addQuestionForm">

<label for="questionText">Question:</label>

<input type="text" id="questionText" required><br><br>

<label for="correctAnswer">Correct Answer:</label>

<input type="text" id="correctAnswer" required><br><br>

<button type="submit">Add Question</button>

</form>

</div>

<div class="form-section">

<h2>Quiz</h2>

<form id="quizForm"></form>

</div>

<button id="submitQuiz">Submit Quiz</button>

<p id="message"></p>

</div>

<script src="script003.js"></script>

</body>

</html>

**CSS Styles**

Next, let's add the CSS styles for a modern, visually appealing design.

/\* styles003.css \*/

body {

font-family: 'Arial', sans-serif;

background: #f4f4f9;

margin: 0;

padding: 0;

}

.container {

max-width: 600px;

margin: 50px auto;

padding: 20px;

background: #fff;

box-shadow: 0 2px 5px rgba(0, 0, 0, 0.1);

border-radius: 8px;

}

h1 {

text-align: center;

color: #333;

}

.form-section {

margin-bottom: 30px;

}

form {

display: flex;

flex-direction: column;

}

label {

margin-bottom: 5px;

font-weight: bold;

color: #555;

}

input[type="text"] {

padding: 10px;

margin-bottom: 15px;

border: 1px solid #ccc;

border-radius: 4px;

}

button {

padding: 10px 15px;

border: none;

border-radius: 4px;

background: #007bff;

color: #fff;

cursor: pointer;

font-size: 16px;

}

button:hover {

background: #0056b3;

}

#message {

text-align: center;

font-size: 18px;

color: #333;

}

.question {

margin-bottom: 20px;

}

**Explanation**

Let's go through each CSS rule line by line to explain its purpose and function.

/\* styles.css \*/

**body {**

**font-family: 'Arial', sans-serif;**

**background: #f4f4f9;**

**margin: 0;**

**padding: 0;**

**}**

* body { ... }: Targets the <body> element of the HTML document.
* font-family: 'Arial', sans-serif;: Sets the font for the body text to Arial, with a fallback to sans-serif if Arial is not available.
* background: #f4f4f9;: Sets the background color of the body to a light gray color (#f4f4f9).
* margin: 0;: Removes the default margin from the body element.
* padding: 0;: Removes the default padding from the body element.

**.container {**

**max-width: 600px;**

**margin: 50px auto;**

**padding: 20px;**

**background: #fff;**

**box-shadow: 0 2px 5px rgba(0, 0, 0, 0.1);**

**border-radius: 8px;**

**}**

* .container { ... }: Targets elements with the class container.
* max-width: 600px;: Limits the maximum width of the container to 600 pixels.
* margin: 50px auto;: Centers the container horizontally and adds a 50-pixel margin at the top and bottom.
* padding: 20px;: Adds a 20-pixel padding inside the container.
* background: #fff;: Sets the background color of the container to white.
* box-shadow: 0 2px 5px rgba(0, 0, 0, 0.1);: Adds a subtle shadow around the container to give it a slight 3D effect.
* border-radius: 8px;: Rounds the corners of the container with an 8-pixel radius.

**h1 {**

**text-align: center;**

**color: #333;**

**}**

* h1 { ... }: Targets <h1> elements.
* text-align: center;: Centers the text horizontally within the <h1> element.
* color: #333;: Sets the text color to a dark gray (#333).

**.form-section {**

**margin-bottom: 30px;**

**}**

* .form-section { ... }: Targets elements with the class form-section.
* margin-bottom: 30px;: Adds a 30-pixel margin below the element.

**form {**

**display: flex;**

**flex-direction: column;**

**}**

* form { ... }: Targets <form> elements.
* display: flex;: Uses Flexbox for layout within the form.
* flex-direction: column;: Arranges the form's child elements in a vertical column.

**label {**

**margin-bottom: 5px;**

**font-weight: bold;**

**color: #555;**

**}**

* label { ... }: Targets <label> elements.
* margin-bottom: 5px;: Adds a 5-pixel margin below each label.
* font-weight: bold;: Makes the label text bold.
* color: #555;: Sets the text color to a medium gray (#555).

**input[type="text"] {**

**padding: 10px;**

**margin-bottom: 15px;**

**border: 1px solid #ccc;**

**border-radius: 4px;**

**}**

* input[type="text"] { ... }: Targets text input fields (<input type="text">).
* padding: 10px;: Adds 10 pixels of padding inside the input field.
* margin-bottom: 15px;: Adds a 15-pixel margin below the input field.
* border: 1px solid #ccc;: Adds a 1-pixel solid border with a light gray color (#ccc).
* border-radius: 4px;: Rounds the corners of the input field with a 4-pixel radius.

**button {**

**padding: 10px 15px;**

**border: none;**

**border-radius: 4px;**

**background: #007bff;**

**color: #fff;**

**cursor: pointer;**

**font-size: 16px;**

**}**

* button { ... }: Targets <button> elements.
* padding: 10px 15px;: Adds 10 pixels of padding at the top and bottom, and 15 pixels at the sides.
* border: none;: Removes any default border.
* border-radius: 4px;: Rounds the corners of the button with a 4-pixel radius.
* background: #007bff;: Sets the background color of the button to a blue color (#007bff).
* color: #fff;: Sets the text color of the button to white.
* cursor: pointer;: Changes the cursor to a pointer when hovering over the button.
* font-size: 16px;: Sets the font size of the button text to 16 pixels.

**button:hover {**

**background: #0056b3;**

**}**

* button:hover { ... }: Targets <button> elements when they are hovered over.
* background: #0056b3;: Changes the background color of the button to a darker blue (#0056b3) when hovered over.

**#message {**

**text-align: center;**

**font-size: 18px;**

**color: #333;**

**}**

* #message { ... }: Targets the element with the id message.
* text-align: center;: Centers the text horizontally within the element.
* font-size: 18px;: Sets the font size of the text to 18 pixels.
* color: #333;: Sets the text color to a dark gray (#333).

**.question {**

**margin-bottom: 20px;**

**}**

* .question { ... }: Targets elements with the class question.
* margin-bottom: 20px;: Adds a 20-pixel margin below the element.

**JavaScript Code**

Finally, let's write the JavaScript code that handles adding questions, taking the quiz, and providing feedback.

document.addEventListener('DOMContentLoaded', function() {

const addQuestionForm = document.getElementById('addQuestionForm');

const quizForm = document.getElementById('quizForm');

const submitQuizButton = document.getElementById('submitQuiz');

const message = document.getElementById('message');

let questions = [];

addQuestionForm.addEventListener('submit', function(event) {

event.preventDefault(); // Prevent the form from submitting the traditional way

const questionText = document.getElementById('questionText').value;

const correctAnswer = document.getElementById('correctAnswer').value;

// Add new question to the questions array

questions.push({ questionText, correctAnswer });

// Clear the form fields

addQuestionForm.reset();

// Render the questions

renderQuestions();

});

function renderQuestions() {

quizForm.innerHTML = ''; // Clear existing questions

// Loop through questions array and create form elements

questions.forEach((question, index) => {

const div = document.createElement('div');

div.className = 'question';

const label = document.createElement('label');

label.textContent = `Question ${index + 1}: ${question.questionText}`;

div.appendChild(label);

const input = document.createElement('input');

input.type = 'text';

input.name = `q${index}`;

input.required = true;

div.appendChild(input);

quizForm.appendChild(div);

});

}

submitQuizButton.addEventListener('click', function(event) {

event.preventDefault(); // Prevent the button from submitting the traditional way

// Check if there are questions to answer

if (questions.length === 0) {

message.textContent = 'Please add questions before taking the quiz.';

return;

}

// Get user's answers

const formData = new FormData(quizForm);

let score = 0;

// Loop through questions and compare answers

questions.forEach((question, index) => {

const userAnswer = formData.get(`q${index}`);

if (userAnswer && userAnswer.toLowerCase() === question.correctAnswer.toLowerCase()) {

score++;

}

});

// Display feedback

const feedback = `You answered ${score} out of ${questions.length} questions correctly.`;

message.textContent = feedback;

});

});

**Explanation:**

Let's go through this JavaScript code line by line.

**document.addEventListener('DOMContentLoaded', function() {**

* Adds an event listener to the document that fires when the DOM content is fully loaded and parsed, ensuring the script runs only after the document is ready.

**const addQuestionForm = document.getElementById('addQuestionForm');**

**const quizForm = document.getElementById('quizForm');**

**const submitQuizButton = document.getElementById('submitQuiz');**

**const message = document.getElementById('message');**

* Retrieves elements by their IDs and assigns them to constants for easy reference:
  + addQuestionForm: The form for adding new questions.
  + quizForm: The form where the quiz questions will be displayed.
  + submitQuizButton: The button to submit the quiz.
  + message: An element to display messages or feedback to the user.

**let questions = [];**

Initializes an empty array to store the questions.

**addQuestionForm.addEventListener('submit', function(event) {**

**event.preventDefault(); // Prevent the form from submitting the traditional way**

**const questionText = document.getElementById('questionText').value;**

**const correctAnswer = document.getElementById('correctAnswer').value;**

**// Add new question to the questions array**

**questions.push({ questionText, correctAnswer });**

**// Clear the form fields**

**addQuestionForm.reset();**

**// Render the questions**

**renderQuestions();**

**});**

* Adds a 'submit' event listener to addQuestionForm that:
  + Prevents the form's default submit action.
  + Retrieves the values of questionText and correctAnswer input fields.
  + Adds the new question (as an object) to the questions array.
  + Resets the form fields to empty.
  + Calls renderQuestions to update the quiz form with the new questions.

**function renderQuestions() {**

**quizForm.innerHTML = ''; // Clear existing questions**

**// Loop through questions array and create form elements**

**questions.forEach((question, index) => {**

**const div = document.createElement('div');**

**div.className = 'question';**

**const label = document.createElement('label');**

**label.textContent = `Question ${index + 1}: ${question.questionText}`;**

**div.appendChild(label);**

**const input = document.createElement('input');**

**input.type = 'text';**

**input.name = `q${index}`;**

**input.required = true;**

**div.appendChild(input);**

**quizForm.appendChild(div);**

**});**

**}**

* Defines the renderQuestions function that:
  + Clears the existing content of quizForm.
  + Iterates over the questions array.
  + For each question:
    - Creates a div element with the class question.
    - Creates a label element with the question text and appends it to the div.
    - Creates an input element for the answer, names it based on its index, sets it as required, and appends it to the div.
    - Appends the div to the quizForm.

**submitQuizButton.addEventListener('click', function(event) {**

**event.preventDefault(); // Prevent the button from submitting the traditional way**

**// Check if there are questions to answer**

**if (questions.length === 0) {**

**message.textContent = 'Please add questions before taking the quiz.';**

**return;**

**}**

**// Get user's answers**

**const formData = new FormData(quizForm);**

**let score = 0;**

**// Loop through questions and compare answers**

**questions.forEach((question, index) => {**

**const userAnswer = formData.get(`q${index}`);**

**if (userAnswer && userAnswer.toLowerCase() === question.correctAnswer.toLowerCase()) {**

**score++;**

**}**

**});**

**// Display feedback**

**const feedback = `You answered ${score} out of ${questions.length} questions correctly.`;**

**message.textContent = feedback;**

**});**

**});**

* Adds a 'click' event listener to submitQuizButton that:
  + Prevents the default button action.
  + Checks if there are any questions; if not, displays a message and exits.
  + Collects the form data using FormData.
  + Initializes a score counter.
  + Iterates over the questions array:
    - Retrieves the user's answer for each question from the form data.
    - Compares it (case-insensitively) to the correct answer.
    - Increments the score if the answers match.
  + Displays feedback with the user's score in the message element.

**Explanation**

1. **HTML Structure:**
   * The HTML includes a form to add new questions and a separate form to take the quiz.
   * A button is provided to submit the quiz and display the results.
2. **CSS Styles:**
   * The CSS styles create a modern and clean design for the quiz application.
   * It includes styles for the body, container, headings, forms, labels, input fields, buttons, and messages.
   * Hover effects and rounded corners enhance the visual appeal.
3. **JavaScript Code:**
   * **Event Listener for Adding Questions:**
     + An event listener on the addQuestionForm handles form submission, preventing the default form submission behavior.
     + The question and correct answer are retrieved from the form fields and added to the questions array.
     + The form fields are cleared, and the renderQuestions function is called to update the quiz form.
   * **Render Questions Function:**
     + This function loops through the questions array and creates form elements for each question.
     + It dynamically creates div elements with labels and input fields for each question and appends them to the quizForm.
   * **Event Listener for Submitting Quiz:**
     + An event listener on the submitQuizButton handles the quiz submission, preventing the default behavior.
     + It checks if there are any questions to answer and displays a message if there are none.
     + It retrieves the user's answers using the FormData object and loops through the questions array to compare the answers.
     + The user's score is calculated, and feedback is displayed based on the number of correct answers.

This example demonstrates how to use loops to dynamically create form elements, handle form submissions, and process user input in a more complex scenario involving a dynamically generated quiz application.

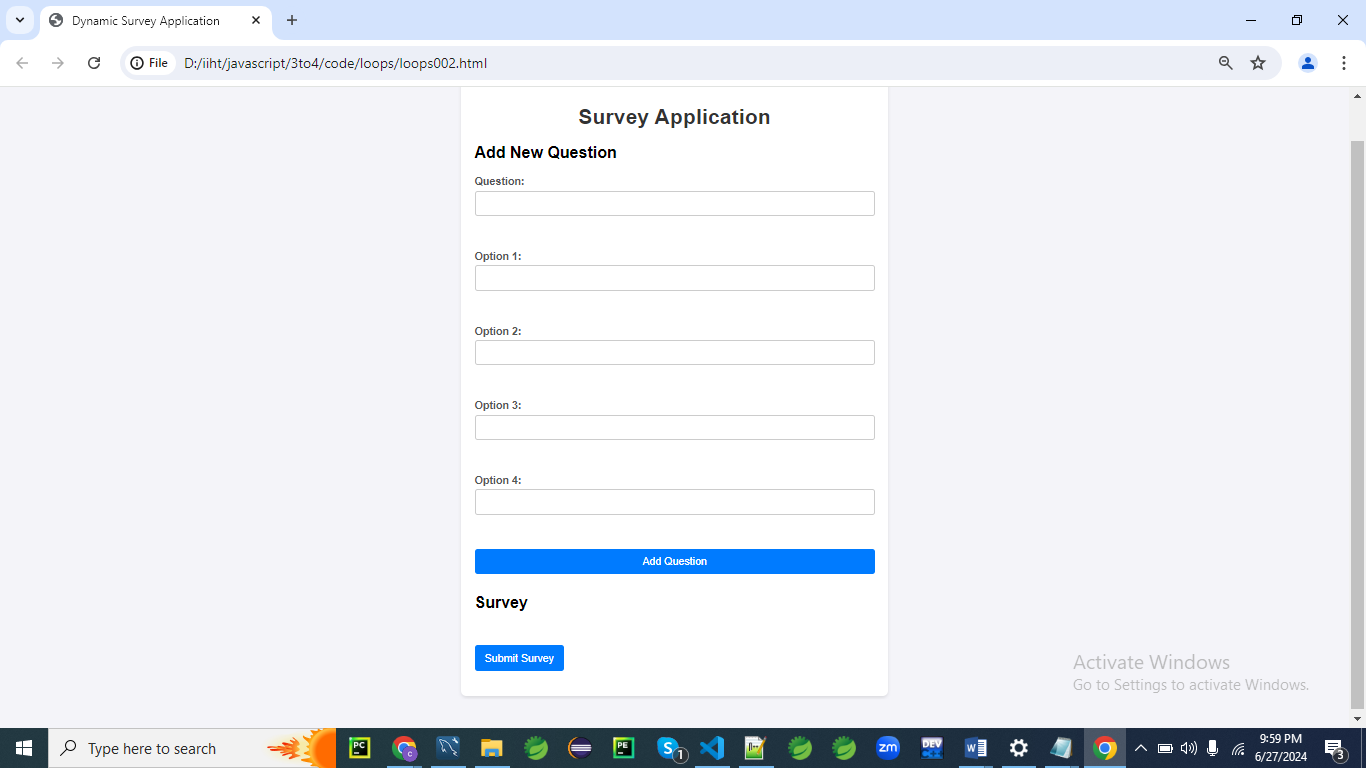
The use of CSS3 styling enhances the visual appeal and user experience.

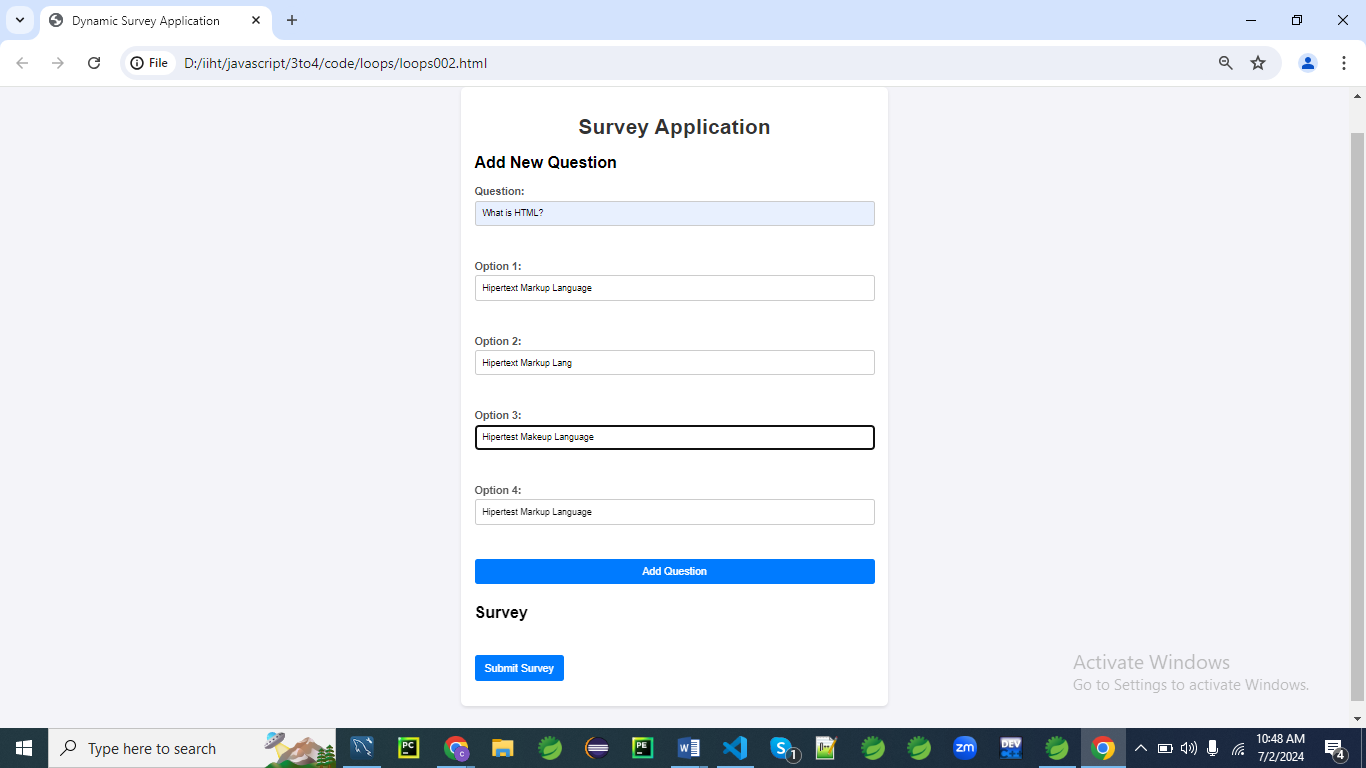
**Example 02**

Let's create a comprehensive example of a survey form where users can add questions dynamically, provide multiple choice answers, and submit their responses.

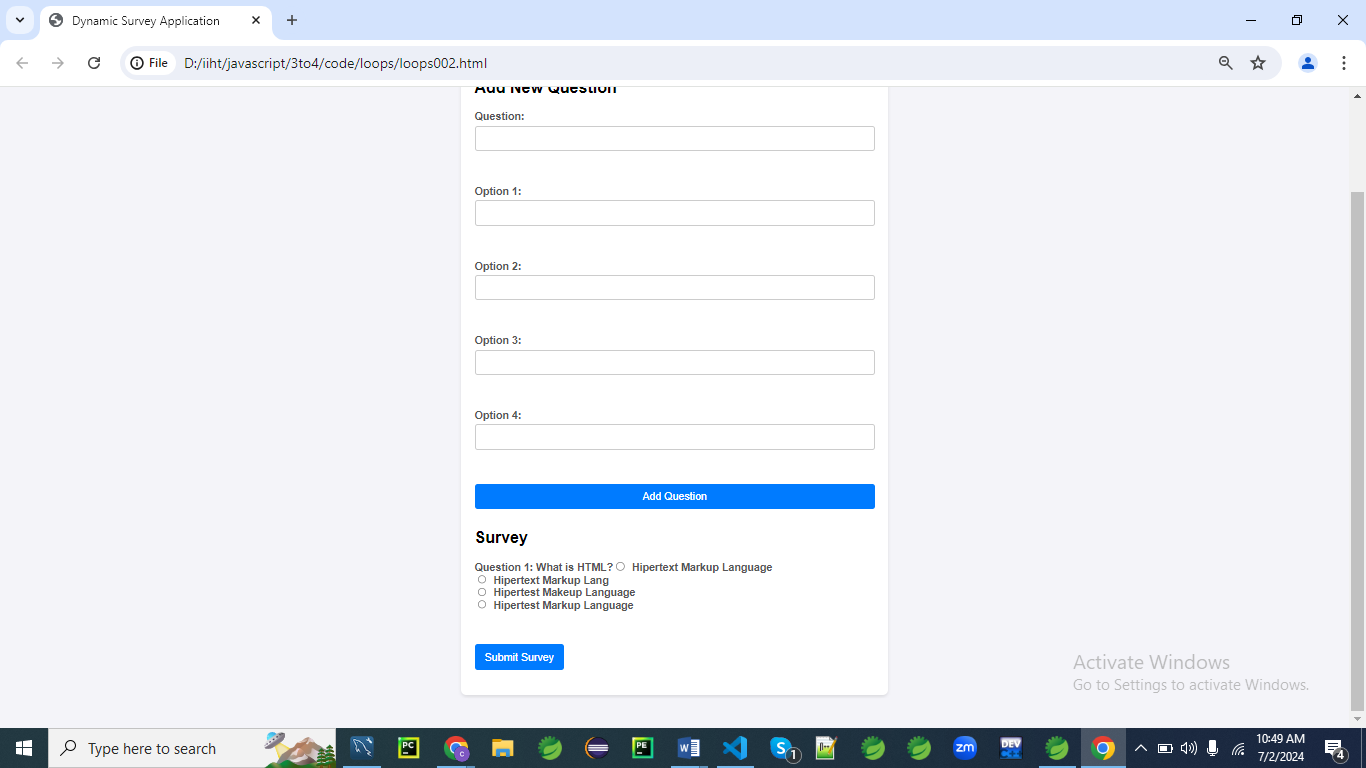
This example will include loops for rendering questions and answers, handling form submissions, and calculating the results.

We'll also enhance the form with modern CSS3 styling.

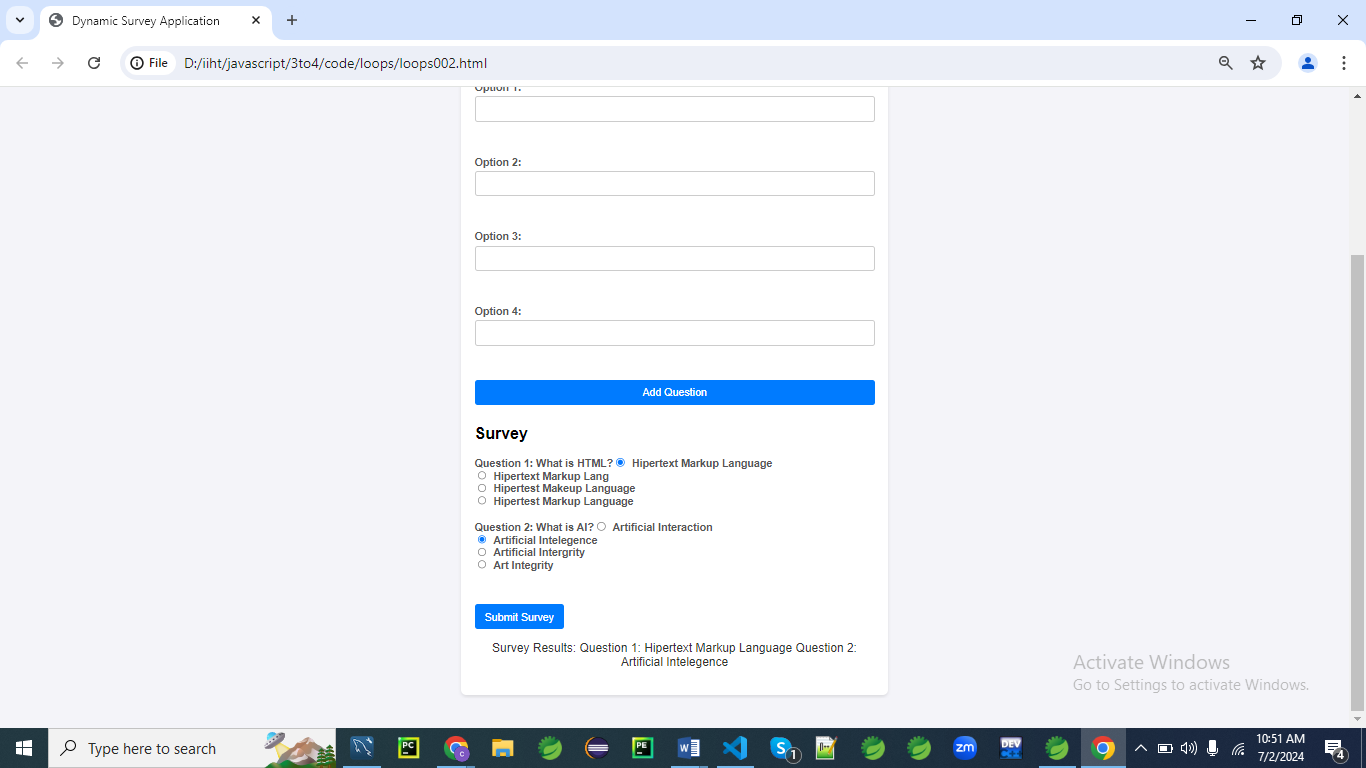




**Add data and click on Add Question**



**Final Output:**



**HTML Form**

First, let's create an HTML structure for the survey application.

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

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

<title>Dynamic Survey Application</title>

<link rel="stylesheet" href="styles.css">

</head>

<body>

<div class="container">

<h1>Survey Application</h1>

<!-- Form to add new questions -->

<div class="form-section">

<h2>Add New Question</h2>

<form id="addQuestionForm">

<label for="questionText">Question:</label>

<input type="text" id="questionText" required><br><br>

<label for="option1">Option 1:</label>

<input type="text" id="option1" required><br><br>

<label for="option2">Option 2:</label>

<input type="text" id="option2" required><br><br>

<label for="option3">Option 3:</label>

<input type="text" id="option3" required><br><br>

<label for="option4">Option 4:</label>

<input type="text" id="option4" required><br><br>

<button type="submit">Add Question</button>

</form>

</div>

<div class="form-section">

<h2>Survey</h2>

<form id="surveyForm"></form>

</div>

<button id="submitSurvey">Submit Survey</button>

<p id="message"></p>

</div>

<script src="script.js"></script>

</body>

</html>

**CSS Styles**

Next, let's add the CSS styles for a modern, visually appealing design.

/\* styles.css \*/

body {

font-family: 'Arial', sans-serif;

background: #f4f4f9;

margin: 0;

padding: 0;

}

.container {

max-width: 600px;

margin: 50px auto;

padding: 20px;

background: #fff;

box-shadow: 0 2px 5px rgba(0, 0, 0, 0.1);

border-radius: 8px;

}

h1 {

text-align: center;

color: #333;

}

.form-section {

margin-bottom: 30px;

}

form {

display: flex;

flex-direction: column;

}

label {

margin-bottom: 5px;

font-weight: bold;

color: #555;

}

input[type="text"] {

padding: 10px;

margin-bottom: 15px;

border: 1px solid #ccc;

border-radius: 4px;

}

input[type="radio"] {

margin-right: 10px;

}

button {

padding: 10px 15px;

border: none;

border-radius: 4px;

background: #007bff;

color: #fff;

cursor: pointer;

font-size: 16px;

}

button:hover {

background: #0056b3;

}

#message {

text-align: center;

font-size: 18px;

color: #333;

}

.question {

margin-bottom: 20px;

}

**Explanation**

Let's go through the CSS code line by line to understand its purpose and effect.

**body {**

**font-family: 'Arial', sans-serif;**

**background: #f4f4f9;**

**margin: 0;**

**padding: 0;**

**}**

* font-family: 'Arial', sans-serif; sets the font of the body to Arial or, if Arial is not available, to a generic sans-serif font.
* background: #f4f4f9; sets the background color of the body to a light gray.
* margin: 0; removes the default margin from the body.
* padding: 0; removes the default padding from the body.

**.container {**

**max-width: 600px;**

**margin: 50px auto;**

**padding: 20px;**

**background: #fff;**

**box-shadow: 0 2px 5px rgba(0, 0, 0, 0.1);**

**border-radius: 8px;**

**}**

* max-width: 600px; restricts the maximum width of the container to 600 pixels.
* margin: 50px auto; centers the container horizontally and adds a 50-pixel margin at the top and bottom.
* padding: 20px; adds 20 pixels of padding inside the container.
* background: #fff; sets the background color of the container to white.
* box-shadow: 0 2px 5px rgba(0, 0, 0, 0.1); adds a subtle shadow around the container to give it a slight 3D effect.
* border-radius: 8px; rounds the corners of the container with an 8-pixel radius.

**h1 {**

**text-align: center;**

**color: #333;**

**}**

* text-align: center; centers the text of all h1 elements.
* color: #333; sets the text color of h1 elements to a dark gray.

**.form-section {**

**margin-bottom: 30px;**

**}**

* margin-bottom: 30px; adds a 30-pixel margin below elements with the form-section class.

**form {**

**display: flex;**

**flex-direction: column;**

**}**

* display: flex; makes all form elements use flexbox layout.
* flex-direction: column; arranges the children of form elements in a column (one below the other).

**label {**

**margin-bottom: 5px;**

**font-weight: bold;**

**color: #555;**

**}**

* margin-bottom: 5px; adds a 5-pixel margin below all label elements.
* font-weight: bold; makes the text of all label elements bold.
* color: #555; sets the text color of all label elements to a medium gray.

**input[type="text"] {**

**padding: 10px;**

**margin-bottom: 15px;**

**border: 1px solid #ccc;**

**border-radius: 4px;**

**}**

* padding: 10px; adds 10 pixels of padding inside all text input elements.
* margin-bottom: 15px; adds a 15-pixel margin below all text input elements.
* border: 1px solid #ccc; sets a 1-pixel solid border with a light gray color around all text input elements.
* border-radius: 4px; rounds the corners of all text input elements with a 4-pixel radius.

**input[type="radio"] {**

**margin-right: 10px;**

**}**

* margin-right: 10px; adds a 10-pixel margin to the right of all radio input elements.

**button {**

**padding: 10px 15px;**

**border: none;**

**border-radius: 4px;**

**background: #007bff;**

**color: #fff;**

**cursor: pointer;**

**font-size: 16px;**

**}**

* padding: 10px 15px; adds 10 pixels of padding vertically and 15 pixels of padding horizontally inside all button elements.
* border: none; removes the default border from all button elements.
* border-radius: 4px; rounds the corners of all button elements with a 4-pixel radius.
* background: #007bff; sets the background color of all button elements to a blue color.
* color: #fff; sets the text color of all button elements to white.
* cursor: pointer; changes the cursor to a pointer when hovering over all button elements.
* font-size: 16px; sets the font size of all button elements to 16 pixels.

**button:hover {**

**background: #0056b3;**

**}**

* background: #0056b3; changes the background color of all button elements to a darker blue when they are hovered over.

**#message {**

**text-align: center;**

**font-size: 18px;**

**color: #333;**

**}**

* text-align: center; centers the text of the element with the ID message.
* font-size: 18px; sets the font size of the message element to 18 pixels.
* color: #333; sets the text color of the message element to dark gray.

**.question {**

**margin-bottom: 20px;**

**}**

* margin-bottom: 20px; adds a 20-pixel margin below elements with the question class.

**JavaScript Code**

Finally, let's write the JavaScript code that handles adding questions, taking the survey, and providing feedback.

document.addEventListener('DOMContentLoaded', function() {

const addQuestionForm = document.getElementById('addQuestionForm');

const surveyForm = document.getElementById('surveyForm');

const submitSurveyButton = document.getElementById('submitSurvey');

const message = document.getElementById('message');

let questions = [];

addQuestionForm.addEventListener('submit', function(event) {

event.preventDefault(); // Prevent the form from submitting the traditional way

const questionText = document.getElementById('questionText').value;

const option1 = document.getElementById('option1').value;

const option2 = document.getElementById('option2').value;

const option3 = document.getElementById('option3').value;

const option4 = document.getElementById('option4').value;

// Add new question to the questions array

questions.push({

questionText,

options: [option1, option2, option3, option4]

});

// Clear the form fields

addQuestionForm.reset();

// Render the questions

renderQuestions();

});

function renderQuestions() {

surveyForm.innerHTML = ''; // Clear existing questions

// Loop through questions array and create form elements

questions.forEach((question, index) => {

const div = document.createElement('div');

div.className = 'question';

const label = document.createElement('label');

label.textContent = `Question ${index + 1}: ${question.questionText}`;

div.appendChild(label);

question.options.forEach((option, optIndex) => {

const optionLabel = document.createElement('label');

const input = document.createElement('input');

input.type = 'radio';

input.name = `q${index}`;

input.value = option;

input.required = true;

optionLabel.appendChild(input);

optionLabel.appendChild(document.createTextNode(option));

div.appendChild(optionLabel);

div.appendChild(document.createElement('br'));

});

surveyForm.appendChild(div);

});

}

submitSurveyButton.addEventListener('click', function(event) {

event.preventDefault(); // Prevent the button from submitting the traditional way

// Check if there are questions to answer

if (questions.length === 0) {

message.textContent = 'Please add questions before taking the survey.';

return;

}

// Get user's answers

const formData = new FormData(surveyForm);

let results = {};

// Loop through questions and record answers

questions.forEach((question, index) => {

const userAnswer = formData.get(`q${index}`);

results[`Question ${index + 1}`] = userAnswer;

});

// Display results

let feedback = 'Survey Results:\n';

for (const [question, answer] of Object.entries(results)) {

feedback += `${question}: ${answer}\n`;

}

message.textContent = feedback;

});

});

**Explanation**

Let's go through the JavaScript code line by line to understand its purpose and functionality.

**document.addEventListener('DOMContentLoaded', function() {**

* This line adds an event listener to the document that waits for the DOM content to be fully loaded before running the enclosed function. This ensures that the JavaScript code runs after the HTML has been completely loaded.

**const addQuestionForm = document.getElementById('addQuestionForm');**

**const surveyForm = document.getElementById('surveyForm');**

**const submitSurveyButton = document.getElementById('submitSurvey');**

**const message = document.getElementById('message');**

* These lines get references to the HTML elements with the IDs addQuestionForm, surveyForm, submitSurvey, and message, and store them in variables for later use.

**let questions = [];**

* This line initializes an empty array questions to store the survey questions and their options.

**addQuestionForm.addEventListener('submit', function(event) {**

* This line adds a submit event listener to the addQuestionForm element. When the form is submitted, the enclosed function will run.

**event.preventDefault(); // Prevent the form from submitting the traditional way**

* This line prevents the default form submission behavior, which would cause the page to reload.

**const questionText = document.getElementById('questionText').value;**

**const option1 = document.getElementById('option1').value;**

**const option2 = document.getElementById('option2').value;**

**const option3 = document.getElementById('option3').value;**

**const option4 = document.getElementById('option4').value;**

* These lines get the values from the input fields with IDs questionText, option1, option2, option3, and option4, and store them in variables.

**// Add new question to the questions array**

**questions.push({**

**questionText,**

**options: [option1, option2, option3, option4]**

**});**

* This line creates a new question object with the text and options, and adds it to the questions array.

**// Clear the form fields**

**addQuestionForm.reset();**

* This line resets the addQuestionForm, clearing all input fields.

**// Render the questions**

**renderQuestions();**

**});**

* This line calls the renderQuestions function to update the survey form with the newly added question.

**function renderQuestions() {**

**surveyForm.innerHTML = ''; // Clear existing questions**

* This line clears any existing questions in the surveyForm.

**// Loop through questions array and create form elements**

**questions.forEach((question, index) => {**

**const div = document.createElement('div');**

**div.className = 'question';**

* This loop iterates over the questions array, creating a new div element for each question and assigning it the class question.

**const label = document.createElement('label');**

**label.textContent = `Question ${index + 1}: ${question.questionText}`;**

**div.appendChild(label);**

* This creates a label element for the question text and appends it to the div.

**question.options.forEach((option, optIndex) => {**

**const optionLabel = document.createElement('label');**

**const input = document.createElement('input');**

**input.type = 'radio';**

**input.name = `q${index}`;**

**input.value = option;**

**input.required = true;**

* This inner loop iterates over the options for each question, creating a label and input element for each option.
* The input elements are of type radio, grouped by name (q${index}) to ensure that only one option can be selected per question.
* The required attribute makes answering each question mandatory.

**optionLabel.appendChild(input);**

**optionLabel.appendChild(document.createTextNode(option));**

**div.appendChild(optionLabel);**

**div.appendChild(document.createElement('br'));**

**});**

* These lines append the input and option text to the optionLabel, then append the optionLabel to the div, and add a line break for layout purposes.

**surveyForm.appendChild(div);**

**});**

**}**

* This line appends the div containing the question and options to the surveyForm.

**submitSurveyButton.addEventListener('click', function(event) {**

**event.preventDefault(); // Prevent the button from submitting the traditional way**

* This line adds a click event listener to the submitSurveyButton. When the button is clicked, the enclosed function will run, and the default form submission behavior will be prevented.

**// Check if there are questions to answer**

**if (questions.length === 0) {**

**message.textContent = 'Please add questions before taking the survey.';**

**return;**

**}**

* This block checks if there are any questions in the questions array. If there are none, it updates the message element with a prompt to add questions and exits the function.

**// Get user's answers**

**const formData = new FormData(surveyForm);**

**let results = {};**

* This line creates a FormData object from the surveyForm to easily access the user's answers. It also initializes an empty object results to store the answers.

**// Loop through questions and record answers**

**questions.forEach((question, index) => {**

**const userAnswer = formData.get(`q${index}`);**

**results[`Question ${index + 1}`] = userAnswer;**

**});**

* This loop iterates over the questions array, retrieves the user's answer for each question from the FormData object, and stores it in the results object.

**// Display results**

**let feedback = 'Survey Results:\n';**

**for (const [question, answer] of Object.entries(results)) {**

**feedback += `${question}: ${answer}\n`;**

**}**

**message.textContent = feedback;**

**});**

**});**

* This block constructs a feedback string from the results object and updates the message element with the survey results.

**Explanation**

1. **HTML Structure:**
   * The HTML includes a form to add new questions and a separate form to take the survey.
   * A button is provided to submit the survey and display the results.
2. **CSS Styles:**
   * The CSS styles create a modern and clean design for the survey application.
   * It includes styles for the body, container, headings, forms, labels, input fields, radio buttons, buttons, and messages.
   * Hover effects and rounded corners enhance the visual appeal.
3. **JavaScript Code:**
   * **Event Listener for Adding Questions:**
     + An event listener on the addQuestionForm handles form submission, preventing the default form submission behavior.
     + The question and options are retrieved from the form fields and added to the questions array.
     + The form fields are cleared, and the renderQuestions function is called to update the survey form.
   * **Render Questions Function:**
     + This function loops through the questions array and creates form elements for each question.
     + It dynamically creates div elements with labels, radio buttons, and options for each question and appends them to the surveyForm.
   * **Event Listener for Submitting Survey:**
     + An event listener on the submitSurveyButton handles the survey submission, preventing the default behavior.
     + It checks if there are any questions to answer and displays a message if there are none.
     + It retrieves the user's answers using the FormData object and loops through the questions array to record the answers.
     + The user's responses are displayed as feedback.

This example demonstrates how to use loops to dynamically create form elements, handle form submissions, and process user input in a more complex scenario involving a dynamically generated survey application. The use of CSS3 styling enhances the visual appeal and user experience.

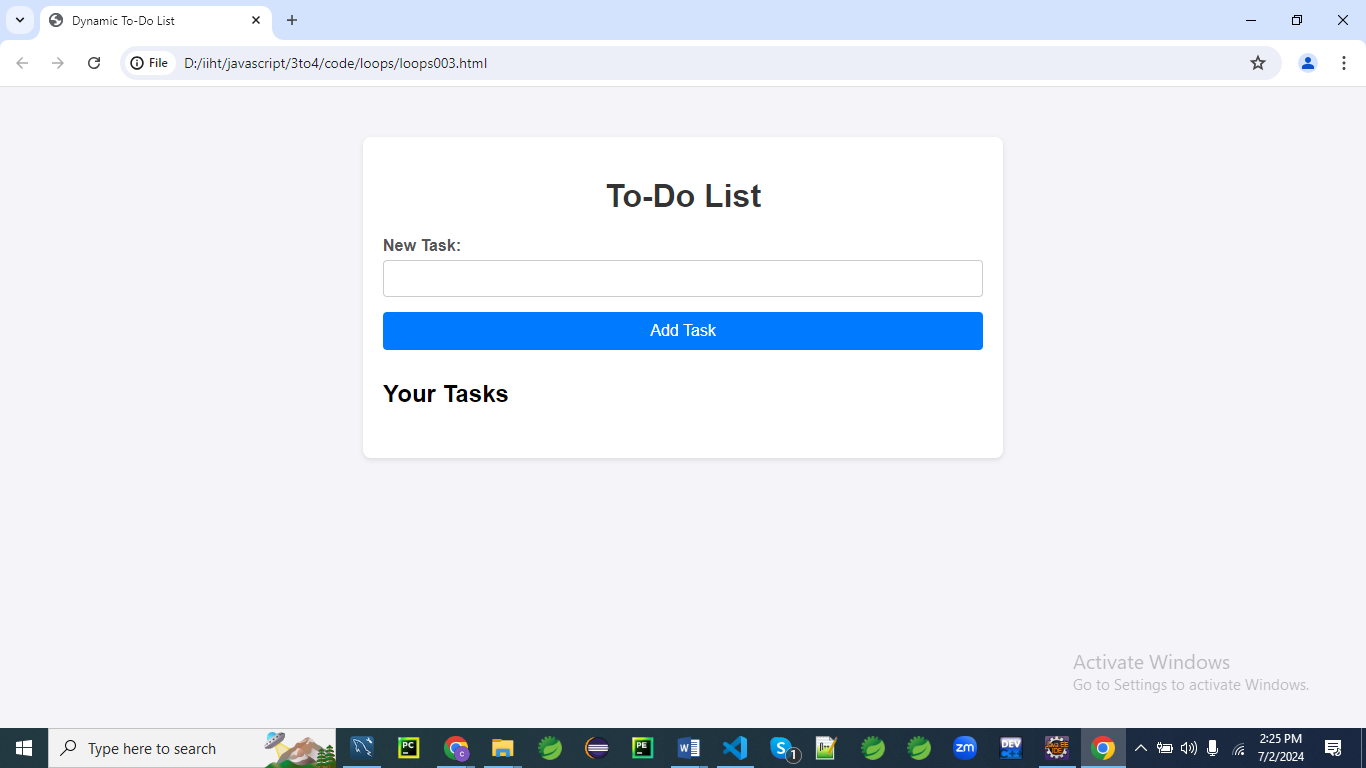
**Example 03**

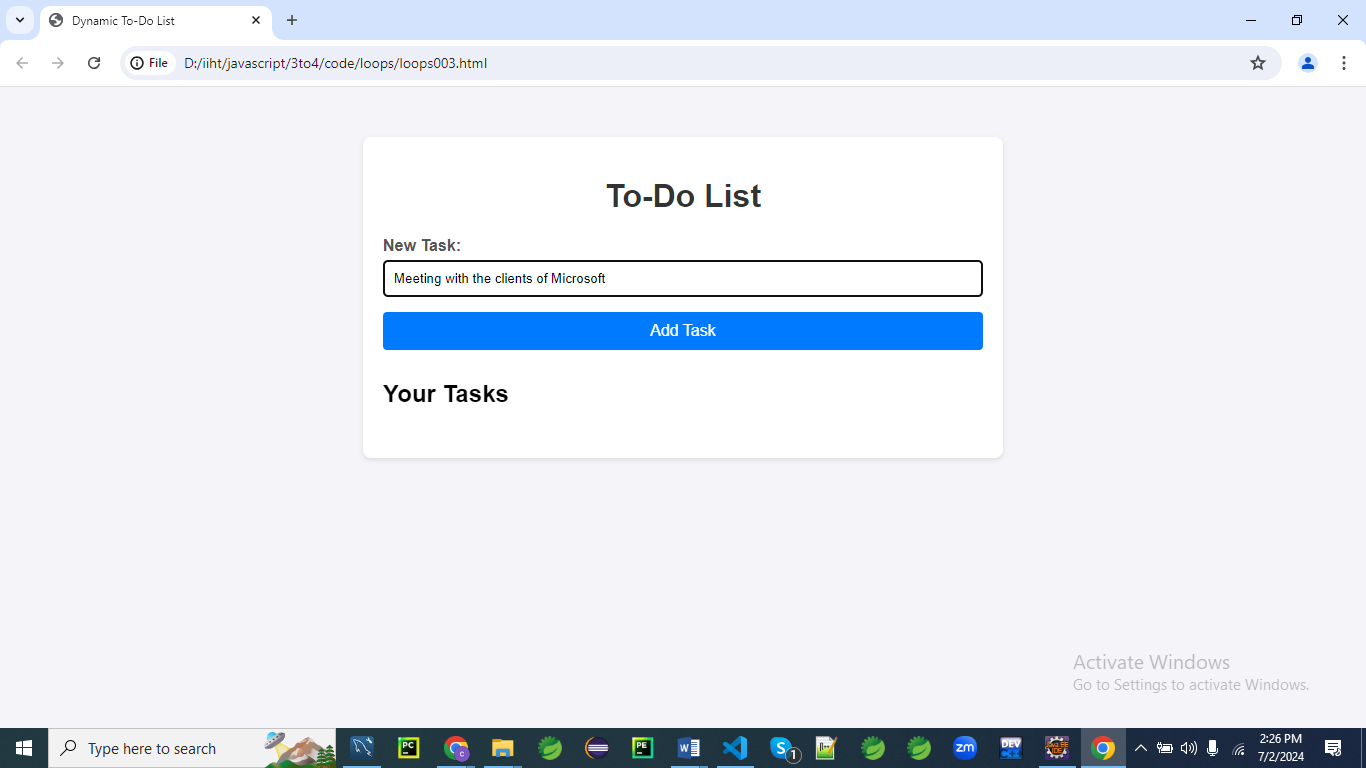
Let's create an example that involves a dynamic to-do list application.

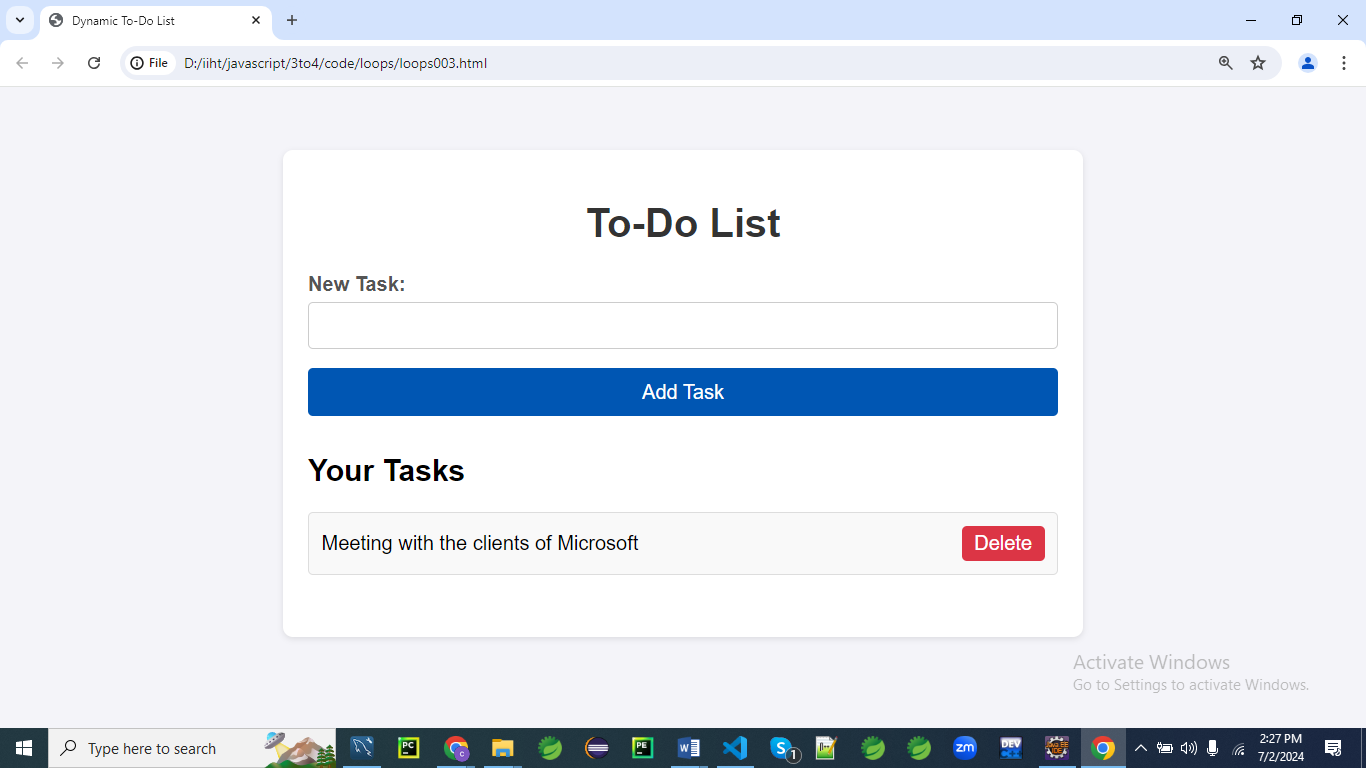
This application will allow users to add tasks, mark them as complete or incomplete, and delete tasks.

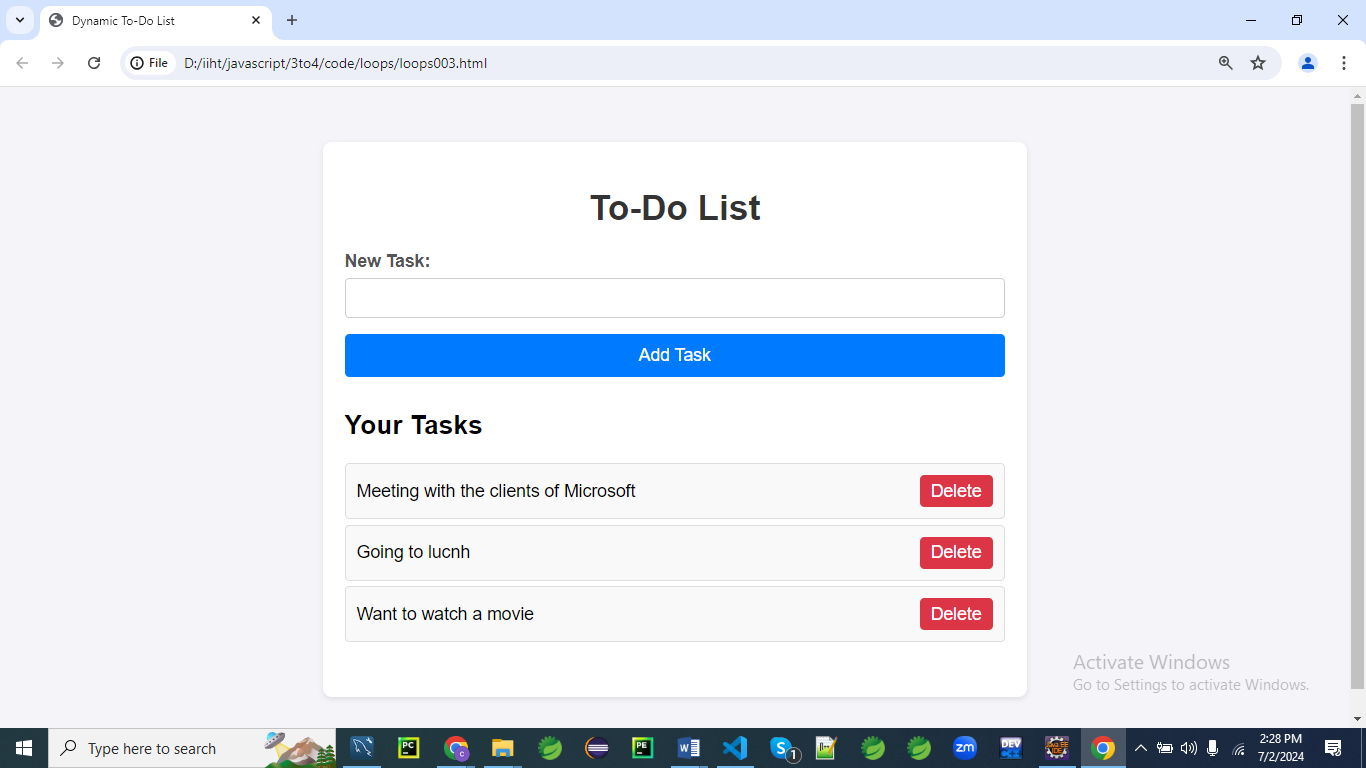
The application will use loops to render tasks and handle user interactions.

Additionally, we'll enhance the form with modern CSS3 styling.

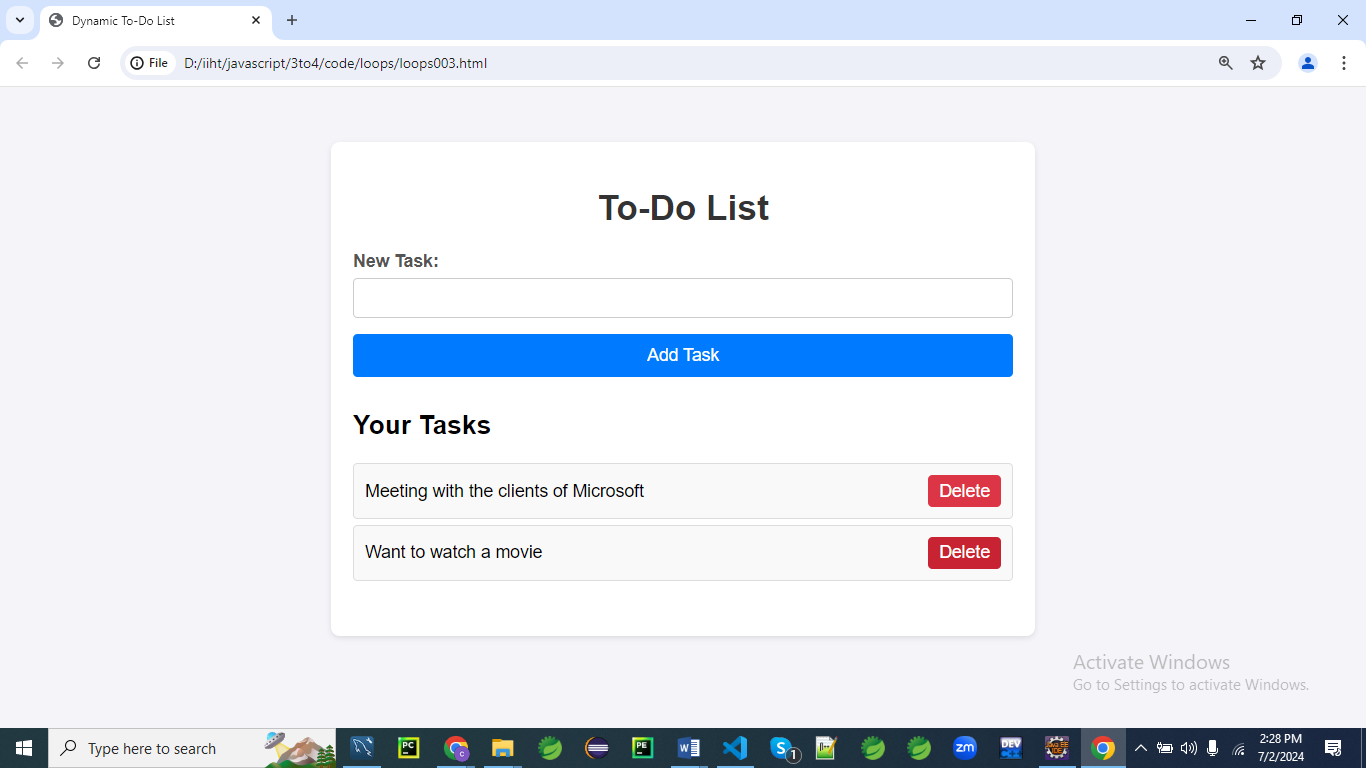








After deleting the option Going to Lunch



**HTML Structure**

First, let's create an HTML structure for the to-do list application.

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

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

<title>Dynamic To-Do List</title>

<link rel="stylesheet" href="styles.css">

</head>

<body>

<div class="container">

<h1>To-Do List</h1>

<!-- Form to add new tasks -->

<div class="form-section">

<form id="addTaskForm">

<label for="taskText">New Task:</label>

<input type="text" id="taskText" required>

<button type="submit">Add Task</button>

</form>

</div>

<div class="form-section">

<h2>Your Tasks</h2>

<ul id="taskList"></ul>

</div>

</div>

<script src="script.js"></script>

</body>

</html>

**CSS Styles**

Next, let's add the CSS styles for a modern, visually appealing design.

/\* styles.css \*/

body {

font-family: 'Arial', sans-serif;

background: #f4f4f9;

margin: 0;

padding: 0;

}

.container {

max-width: 600px;

margin: 50px auto;

padding: 20px;

background: #fff;

box-shadow: 0 2px 5px rgba(0, 0, 0, 0.1);

border-radius: 8px;

}

h1 {

text-align: center;

color: #333;

}

.form-section {

margin-bottom: 30px;

}

form {

display: flex;

flex-direction: column;

}

label {

margin-bottom: 5px;

font-weight: bold;

color: #555;

}

input[type="text"] {

padding: 10px;

margin-bottom: 15px;

border: 1px solid #ccc;

border-radius: 4px;

}

button {

padding: 10px 15px;

border: none;

border-radius: 4px;

background: #007bff;

color: #fff;

cursor: pointer;

font-size: 16px;

}

button:hover {

background: #0056b3;

}

ul {

list-style-type: none;

padding: 0;

}

li {

padding: 10px;

margin: 5px 0;

background: #f9f9f9;

border: 1px solid #ddd;

border-radius: 4px;

display: flex;

justify-content: space-between;

align-items: center;

}

li.completed {

text-decoration: line-through;

color: #888;

}

li button {

background: #dc3545;

border: none;

color: white;

cursor: pointer;

padding: 5px 10px;

border-radius: 4px;

}

li button:hover {

background: #c82333;

}

**Explanation**

Let's go through the CSS code line by line to understand its purpose and functionality.

**body {**

**font-family: 'Arial', sans-serif;**

**background: #f4f4f9;**

**margin: 0;**

**padding: 0;**

**}**

* font-family: 'Arial', sans-serif; sets the font family to Arial, with sans-serif as a fallback.
* background: #f4f4f9; sets the background color of the body to a light grey (#f4f4f9).
* margin: 0; removes any default margin around the body.
* padding: 0; removes any default padding within the body.

**.container {**

**max-width: 600px;**

**margin: 50px auto;**

**padding: 20px;**

**background: #fff;**

**box-shadow: 0 2px 5px rgba(0, 0, 0, 0.1);**

**border-radius: 8px;**

**}**

* max-width: 600px; sets the maximum width of the container to 600 pixels.
* margin: 50px auto; centers the container horizontally and adds 50 pixels of margin above and below it.
* padding: 20px; adds 20 pixels of padding inside the container.
* background: #fff; sets the background color of the container to white.
* box-shadow: 0 2px 5px rgba(0, 0, 0, 0.1); adds a subtle shadow to the container for depth.
* border-radius: 8px; rounds the corners of the container with an 8-pixel radius.

**h1 {**

**text-align: center;**

**color: #333;**

**}**

* text-align: center; centers the text within the <h1> element.
* color: #333; sets the text color to a dark grey (#333).

**.form-section {**

**margin-bottom: 30px;**

**}**

* margin-bottom: 30px; adds 30 pixels of margin below each .form-section element.

**form {**

**display: flex;**

**flex-direction: column;**

**}**

* display: flex; makes the form a flex container.
* flex-direction: column; stacks the form's children vertically.

**label {**

**margin-bottom: 5px;**

**font-weight: bold;**

**color: #555;**

**}**

* margin-bottom: 5px; adds 5 pixels of margin below each label.
* font-weight: bold; makes the label text bold.
* color: #555; sets the label text color to a medium grey (#555).

**input[type="text"] {**

**padding: 10px;**

**margin-bottom: 15px;**

**border: 1px solid #ccc;**

**border-radius: 4px;**

**}**

* padding: 10px; adds 10 pixels of padding inside the text input.
* margin-bottom: 15px; adds 15 pixels of margin below the text input.
* border: 1px solid #ccc; gives the text input a 1-pixel border with a light grey color (#ccc).
* border-radius: 4px; rounds the corners of the text input with a 4-pixel radius.

**button {**

**padding: 10px 15px;**

**border: none;**

**border-radius: 4px;**

**background: #007bff;**

**color: #fff;**

**cursor: pointer;**

**font-size: 16px;**

**}**

* padding: 10px 15px; adds 10 pixels of vertical and 15 pixels of horizontal padding inside the button.
* border: none; removes the default border from the button.
* border-radius: 4px; rounds the corners of the button with a 4-pixel radius.
* background: #007bff; sets the button's background color to a blue shade (#007bff).
* color: #fff; sets the button's text color to white.
* cursor: pointer; changes the cursor to a pointer when hovering over the button.
* font-size: 16px; sets the font size of the button's text to 16 pixels.

**button:hover {**

**background: #0056b3;**

**}**

* background: #0056b3; changes the button's background color to a darker blue (#0056b3) when hovered over.

**ul {**

**list-style-type: none;**

**padding: 0;**

**}**

* list-style-type: none; removes the default list bullets.
* padding: 0; removes the default padding from the unordered list.

**li {**

**padding: 10px;**

**margin: 5px 0;**

**background: #f9f9f9;**

**border: 1px solid #ddd;**

**border-radius: 4px;**

**display: flex;**

**justify-content: space-between;**

**align-items: center;**

**}**

* padding: 10px; adds 10 pixels of padding inside each list item.
* margin: 5px 0; adds 5 pixels of margin above and below each list item.
* background: #f9f9f9; sets the list item's background color to a very light grey (#f9f9f9).
* border: 1px solid #ddd; gives the list item a 1-pixel border with a light grey color (#ddd).
* border-radius: 4px; rounds the corners of the list item with a 4-pixel radius.
* display: flex; makes each list item a flex container.
* justify-content: space-between; spaces the child elements of the list item evenly with space between them.
* align-items: center; vertically centers the child elements of the list item.

**li.completed {**

**text-decoration: line-through;**

**color: #888;**

**}**

* text-decoration: line-through; adds a line through the text of completed list items.
* color: #888; sets the text color of completed list items to a light grey (#888).

**li button {**

**background: #dc3545;**

**border: none;**

**color: white;**

**cursor: pointer;**

**padding: 5px 10px;**

**border-radius: 4px;**

**}**

* background: #dc3545; sets the button's background color to a red shade (#dc3545).
* border: none; removes the default border from the button.
* color: white; sets the button's text color to white.
* cursor: pointer; changes the cursor to a pointer when hovering over the button.
* padding: 5px 10px; adds 5 pixels of vertical and 10 pixels of horizontal padding inside the button.
* border-radius: 4px; rounds the corners of the button with a 4-pixel radius.

**li button:hover {**

**background: #c82333;**

**}**

* background: #c82333; changes the button's background color to a darker red (#c82333) when hovered over.

**JavaScript Code**

Finally, let's write the JavaScript code that handles adding tasks, marking tasks as complete or incomplete, and deleting tasks.

document.addEventListener('DOMContentLoaded', function() {

const addTaskForm = document.getElementById('addTaskForm');

const taskList = document.getElementById('taskList');

let tasks = [];

addTaskForm.addEventListener('submit', function(event) {

event.preventDefault(); // Prevent the form from submitting the traditional way

const taskText = document.getElementById('taskText').value;

// Add new task to the tasks array

tasks.push({ text: taskText, completed: false });

// Clear the form field

addTaskForm.reset();

// Render the tasks

renderTasks();

});

function renderTasks() {

taskList.innerHTML = ''; // Clear existing tasks

// Loop through tasks array and create list elements

**tasks.forEach**((task, index) => {

const li = document.createElement('li');

li.className = task.completed ? 'completed' : '';

const span = document.createElement('span');

span.textContent = task.text;

span.addEventListener('click', () => toggleTask(index));

li.appendChild(span);

const deleteButton = document.createElement('button');

deleteButton.textContent = 'Delete';

deleteButton.addEventListener('click', () => deleteTask(index));

li.appendChild(deleteButton);

taskList.appendChild(li);

});

}

function toggleTask(index) {

tasks[index].completed = !tasks[index].completed;

renderTasks();

}

function deleteTask(index) {

tasks.splice(index, 1);

renderTasks();

}

});

**Explanation**

Let's break down the JavaScript code line by line with detailed explanations.

**document.addEventListener('DOMContentLoaded', function() {**

1. This line adds an event listener to the document that listens for the DOMContentLoaded event.
2. The DOMContentLoaded event is fired when the initial HTML document has been completely loaded and parsed, without waiting for stylesheets, images, and subframes to finish loading.
3. The function passed as a callback will be executed once the DOM is fully loaded, ensuring that all DOM elements are available for manipulation.

**const addTaskForm = document.getElementById('addTaskForm');**

**const taskList = document.getElementById('taskList');**

1. These lines select the form element with the ID addTaskForm and assign it to the constant addTaskForm.
2. Similarly, the unordered list element with the ID taskList is selected and assigned to the constant taskList.
3. These constants will be used to interact with the form and list elements in the DOM.

**let tasks = [];**

1. This line initializes an empty array named tasks.
2. The tasks array will be used to store the list of task objects.
3. Each task object will contain information about the task text and its completion status.

**addTaskForm.addEventListener('submit', function(event) {**

1. This line adds an event listener to the addTaskForm that listens for the submit event.
2. When the form is submitted, the provided callback function will be executed.
3. The submit event occurs when the form is submitted, either by clicking the submit button or pressing Enter.

**event.preventDefault(); // Prevent the form from submitting the traditional way**

1. The preventDefault method is called on the event object to prevent the default form submission behavior.
2. Preventing the default behavior allows us to handle the form submission using JavaScript instead of refreshing the page.
3. This is crucial for adding tasks dynamically without reloading the page.

const taskText = document.getElementById('taskText').value;

1. This line selects the input element with the ID taskText and retrieves its value.
2. The value of the input field (i.e., the task text entered by the user) is stored in the taskText variable.
3. This value will be used to create a new task object in the next step.

**// Add new task to the tasks array**

**tasks.push({ text: taskText, completed: false });**

1. A new task object is created with the text property set to the value of taskText and the completed property set to false.
2. The new task object is then added to the tasks array using the push method.
3. This effectively adds a new task to the list of tasks.

**// Clear the form field**

**addTaskForm.reset();**

1. The reset method is called on the addTaskForm to clear all its input fields.
2. This ensures that the input field is empty and ready for a new task to be entered.
3. This step improves the user experience by resetting the form after a task is added.

**// Render the tasks**

**renderTasks();**

1. The renderTasks function is called to update the task list displayed on the page.
2. This function will dynamically create and display the list items for each task in the tasks array.
3. By calling renderTasks, the new task is immediately visible to the user.

**function renderTasks() {**

**taskList.innerHTML = ''; // Clear existing tasks**

1. The renderTasks function starts by clearing the existing content of the taskList element.
2. Setting taskList.innerHTML to an empty string removes all current child elements (i.e., list items) from the task list.
3. This ensures that the task list is refreshed and avoids duplicating tasks when the list is re-rendered.

**// Loop through tasks array and create list elements**

**tasks.forEach((task, index) => {**

**const li = document.createElement('li');**

**li.className = task.completed ? 'completed' : '';**

1. The forEach method is used to iterate over each task in the tasks array.
2. For each task, a new <li> element is created using document.createElement('li').
3. The className of the <li> element is set to 'completed' if the task is completed, or an empty string otherwise.

**const span = document.createElement('span');**

**span.textContent = task.text;**

**span.addEventListener('click', () => toggleTask(index));**

**li.appendChild(span);**

1. A new <span> element is created to hold the task text.
2. The textContent property of the <span> element is set to the task's text.
3. An event listener is added to the <span> element that calls the toggleTask function when the <span> is clicked. This will toggle the task's completion status.

**const deleteButton = document.createElement('button');**

**deleteButton.textContent = 'Delete';**

**deleteButton.addEventListener('click', () => deleteTask(index));**

**li.appendChild(deleteButton);**

1. A new <button> element is created with the text content 'Delete'.
2. An event listener is added to the deleteButton that calls the deleteTask function when the button is clicked. This will delete the task.
3. The deleteButton is appended to the <li> element.

**taskList.appendChild(li);**

**});**

**}**

1. The <li> element, which now contains the task text and delete button, is appended to the taskList element.
2. This process is repeated for each task in the tasks array, resulting in a complete list of tasks displayed in the DOM.
3. The renderTasks function ends after all tasks have been added to the taskList.

**function toggleTask(index) {**

**tasks[index].completed = !tasks[index].completed;**

**renderTasks();**

**}**

1. The toggleTask function takes the index of a task as an argument.
2. It toggles the completed property of the specified task by setting it to the opposite of its current value.
3. The renderTasks function is called to update the task list display.

**function deleteTask(index) {**

**tasks.splice(index, 1);**

**renderTasks();**

**}**

1. The deleteTask function takes the index of a task as an argument.
2. It removes the specified task from the tasks array using the splice method.
3. The renderTasks function is called to update the task list display.

**});**

1. This line closes the callback function passed to the DOMContentLoaded event listener.
2. It ensures all the defined functions and event listeners are executed only after the DOM is fully loaded.
3. The script ends, encapsulating all the functionality within the DOMContentLoaded event listener.

**Explanation**

1. **HTML Structure:**
   * The HTML includes a form to add new tasks and an unordered list to display the tasks.
   * A button is provided to submit the new task.
2. **CSS Styles:**
   * The CSS styles create a modern and clean design for the to-do list application.
   * It includes styles for the body, container, headings, forms, labels, input fields, buttons, list items, and task completion states.
   * Hover effects and rounded corners enhance the visual appeal.
3. **JavaScript Code:**
   * **Event Listener for Adding Tasks:**
     + An event listener on the addTaskForm handles form submission, preventing the default form submission behavior.
     + The task text is retrieved from the form field and added to the tasks array.
     + The form field is cleared, and the renderTasks function is called to update the task list.
   * **Render Tasks Function:**
     + This function loops through the tasks array and creates list elements for each task.
     + It dynamically creates li elements with spans and buttons for each task and appends them to the taskList.
   * **Toggle Task Function:**
     + This function toggles the completion state of a task when its span is clicked.
     + It updates the completed property of the task and re-renders the task list.
   * **Delete Task Function:**
     + This function deletes a task when its delete button is clicked.
     + It removes the task from the tasks array and re-renders the task list.

This example demonstrates how to use loops to dynamically create list elements, handle form submissions, and process user input in a more complex scenario involving a dynamically generated to-do list application. The use of CSS3 styling enhances the visual appeal and user experience.