Events and Event Listeners

Introduction to Events

An **event** in JavaScript refers to an action that occurs as a result of user interaction or system processes. Examples of events include clicking a button, typing in a text field, hovering over an element, and loading a webpage.

JavaScript enables developers to handle these events using **event listeners**, which allow us to execute specific code when an event occurs.

Types of Events

There are numerous types of events in JavaScript, but they can be categorized into:

1. **Mouse Events**
    - click - Triggered when an element is clicked.
    - dblclick - Triggered when an element is double-clicked.
    - mouseover - Triggered when the mouse enters an element.
    - mouseout - Triggered when the mouse leaves an element.
    - mousemove - Triggered when the mouse moves over an element.
2. **Keyboard Events**
    - keydown - Triggered when a key is pressed.
    - keyup - Triggered when a key is released.
    - keypress - Triggered when a key is pressed down (deprecated in modern browsers).
3. **Form Events**
    - submit - Triggered when a form is submitted.
    - change - Triggered when an input value changes.
    - focus - Triggered when an input field gains focus.
    - blur - Triggered when an input field loses focus.
4. **Window Events**
    - load - Triggered when a page or an image has fully loaded.
    - resize - Triggered when the browser window is resized.
    - scroll - Triggered when the user scrolls.

Event Listeners

An **event listener** is a function that waits for a specific event to occur on an element and then executes a predefined action. Event listeners can be attached to any DOM element.

Adding an Event Listener

The addEventListener() method is used to attach an event to an element.

Syntax:

```
element.addEventListener("event", function, useCapture);
```

- event: The name of the event (e.g., "click", "mouseover").
- function: The function to be executed when the event occurs.
- useCapture (optional): A boolean that specifies whether to use event capturing or bubbling (default is false, meaning bubbling phase).

Example: Click Event Listener

```
document.getElementById("myButton").addEventListener("click", function() {
  alert("Button clicked!");
});
```

Example: Mouseover Event Listener

```
document.getElementById("myDiv").addEventListener("mouseover", function() {
  this.style.backgroundColor = "yellow";
});
```

Removing an Event Listener

An event listener can be removed using the removeEventListener() method.

Example:

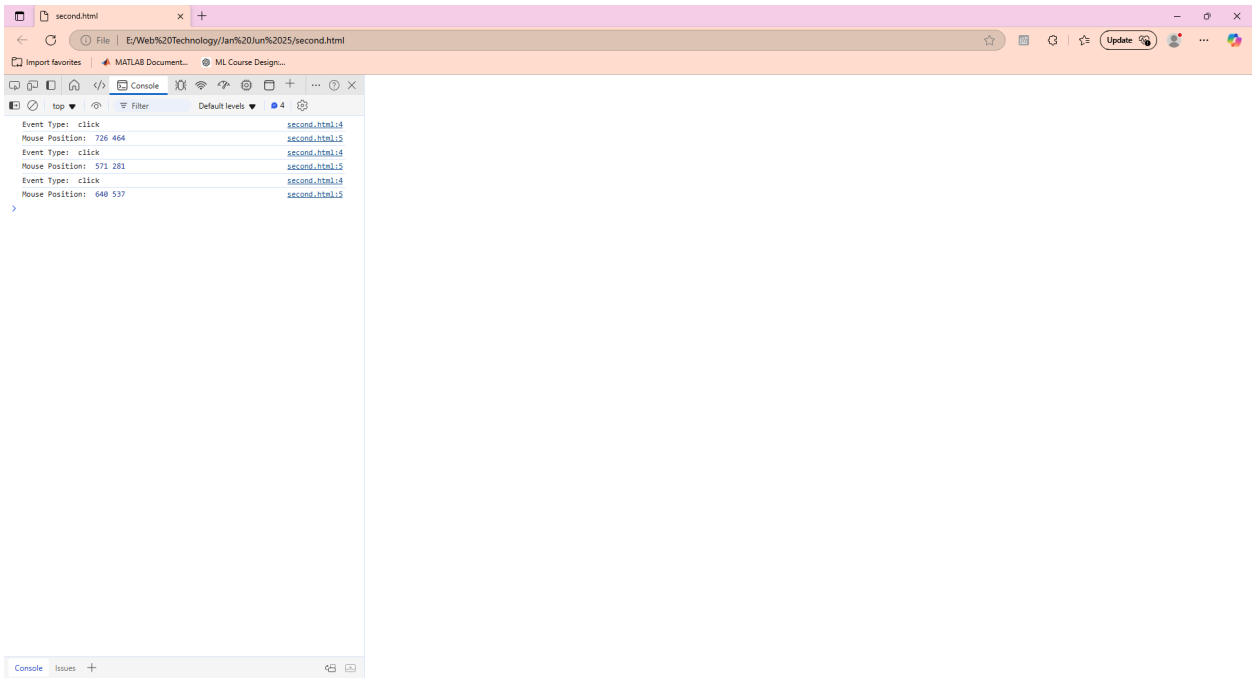
```
function showMessage() {
  alert("Event listener removed!");
}
document.getElementById("myButton").addEventListener("click", showMessage);
document.getElementById("myButton").removeEventListener("click", showMessage);
```

Event Object

When an event occurs, an **event object** is automatically passed to the event handler. This object contains useful information about the event.

Example: Accessing Event Properties

```
document.addEventListener("click", function(event) {
  console.log("Event Type: ", event.type);
  console.log("Mouse Position: ", event.clientX, event.clientY);
});
```



Event Propagation (Bubbling & Capturing)

Event Bubbling

- The event starts from the target element and bubbles up to the root.
- Default behavior in JavaScript.

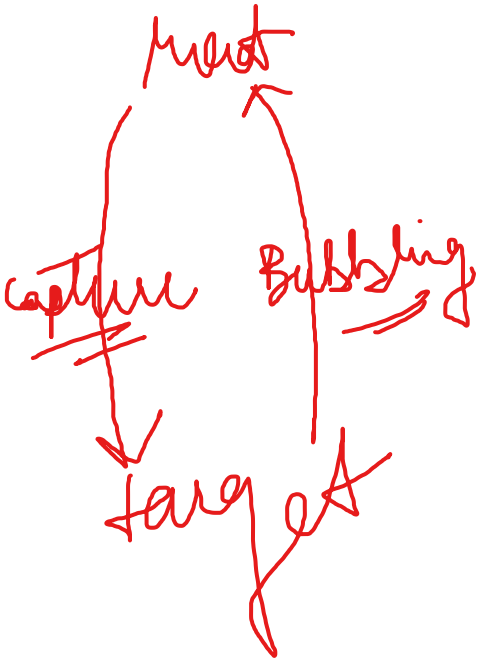
Event Capturing

- The event starts from the root and propagates down to the target.
- Enabled by setting useCapture to true in addEventListener().

Example: Bubbling vs Capturing

```
document.getElementById("parent").addEventListener("click", function() {
  alert("Parent clicked!");
}, true); // Capturing phase

document.getElementById("child").addEventListener("click", function() {
  alert("Child clicked!");
}, false); // Bubbling phase
```



Preventing Default Behavior

Some events have default behaviors, like submitting a form or following a link. You can prevent these using event.preventDefault().

Example: Prevent Form Submission

```
document.getElementById("myForm").addEventListener("submit", function(event) {
  event.preventDefault(); // Stops form from submitting
  alert("Form submission prevented!");
});
```

ES6 Features (let, const, arrow functions, template literals)

Introduction to ES6

ECMAScript 6 (ES6), also known as ECMAScript 2015, introduced several improvements to JavaScript, making the language more powerful and easier to work with. Some of the most important features include let and const, arrow functions, and template literals.

1. let and const

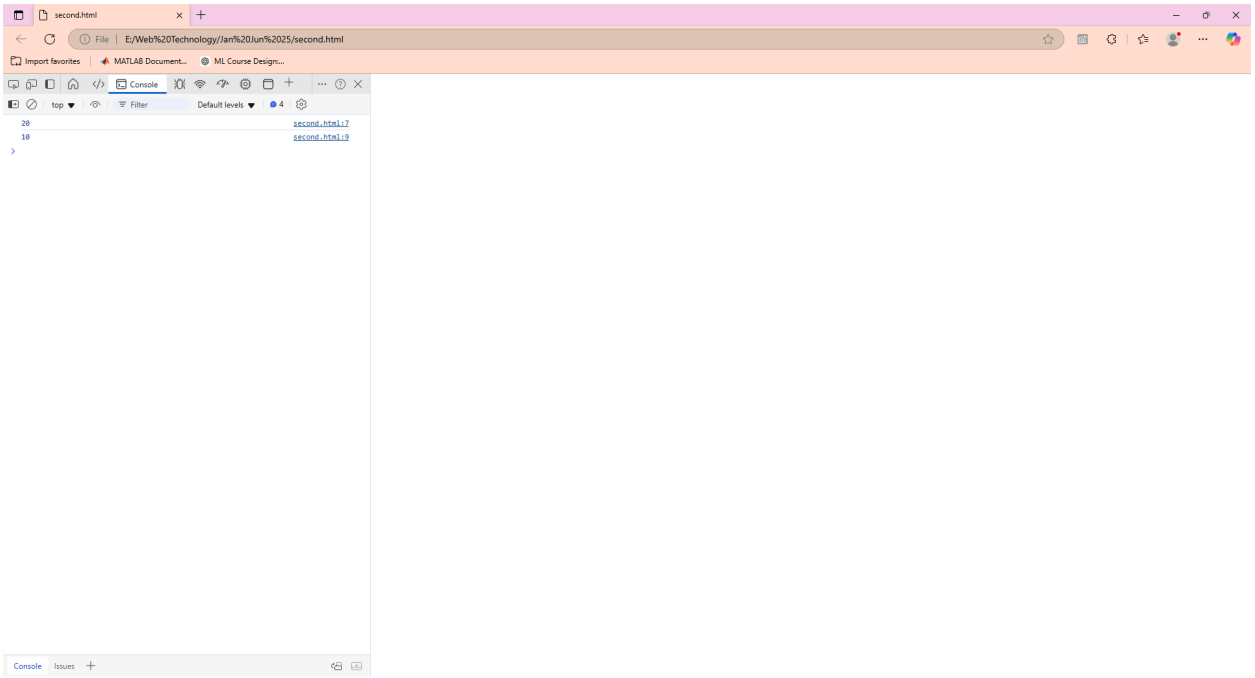
Basics JS

let

- let allows block-scoped variable declarations.
- Variables declared with let can be reassigned but not redeclared in the same scope.

Example:

```
let x = 10;
if (true) {
  let x = 20;
  console.log(x); // 20 (block scope)
}
console.log(x); // 10 (outer scope)
```



const

- const also allows block-scoped variable declarations.
- Variables declared with const cannot be reassigned after initialization.

Example:

```
const pi = 3.14;
// pi = 3.1415; // Error: Assignment to constant variable
```

2. Arrow Functions

- Arrow functions provide a more concise way to write function expressions.
- They use the => syntax and do not bind their own this, making them useful in callback functions.

Example:

```
const add = (a, b) => a + b;
console.log(add(5, 10)); // 15
```

Arrow function without parameters:

```
const greet = () => console.log("Hello, World!");
greet(); // Hello, World!
```

Arrow function with a single parameter (parentheses can be omitted):

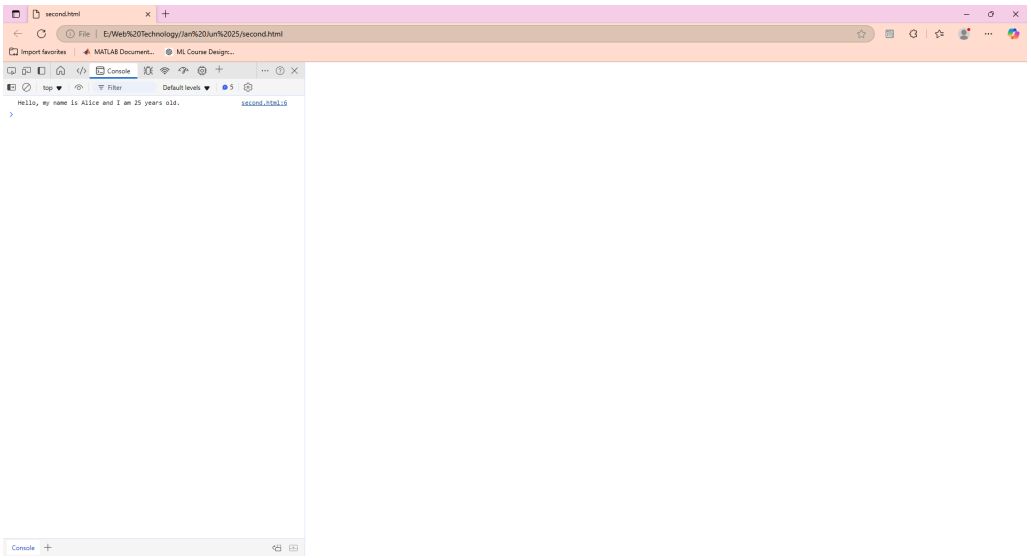
```
const square = x => x * x;
console.log(square(4)); // 16
```

3. Template Literals

- Template literals provide a cleaner way to handle string interpolation.
- They use backticks (`) instead of quotes and allow variables inside the string using ${}.

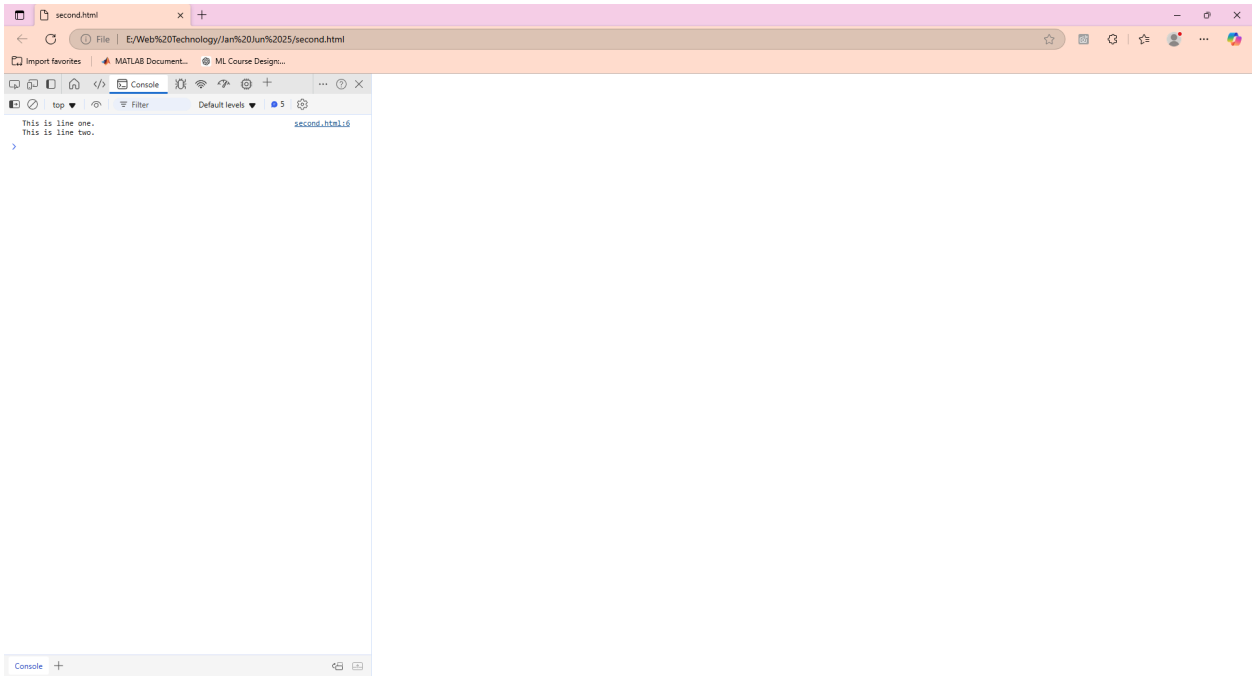
Example:

```
let name = "Alice";
let age = 25;
console.log(`Hello, my name is ${name} and I am ${age} years old.`);
```



Multiline Strings with Template Literals:

```
let multiLine = `This is line one.
This is line two.`;
console.log(multiLine);
```



Fetch API and JSON

1. Introduction to Fetch API

The Fetch API provides a modern, flexible way to make HTTP requests. It is a replacement for older techniques like XMLHttpRequest. The Fetch API returns a Promise, which allows handling network requests asynchronously.

Basic Syntax

```
fetch(url)
  .then(response => response.json())
  .then(data => console.log(data))
  .catch(error => console.error('Error:', error));
```

2. Fetching Data from an API

Example: GET Request

```
fetch('https://jsonplaceholder.typicode.com/posts/1')
 .then(response => response.json())  // Convert response to JSON
 .then(data => console.log(data))    // Handle data
 .catch(error => console.error('Error:', error)); // Handle errors
```

Explanation

- fetch(url) makes an HTTP request to the specified URL.
- .then(response => response.json()) converts the response to a JSON object.
- .then(data => console.log(data)) processes the returned data.
- .catch(error => console.error(error)) catches any errors.

3. Sending Data Using Fetch API

The Fetch API allows sending data using various HTTP methods such as POST, PUT, and DELETE.

Example: POST Request

```
fetch('https://jsonplaceholder.typicode.com/posts', {
 method: 'POST',
 headers: {
  'Content-Type': 'application/json'
 },
 body: JSON.stringify({
  title: 'New Post',
  body: 'This is a new post',
  userId: 1
 })
})
.then(response => response.json())
.then(data => console.log(data))
.catch(error => console.error('Error:', error));
```

Explanation

- The method is set to 'POST', indicating we are sending data.

- The headers define the content type as application/json.
- The body contains the JSON data converted to a string using JSON.stringify().
- The response is processed similarly to the GET request.

4. Understanding JSON

JSON (JavaScript Object Notation) is a lightweight format used to store and exchange data. It is text-based and language-independent.

Example of JSON Data

```
{
 "name": "John",
 "age": 30,
 "city": "New York"
}
```

Why Use JSON?

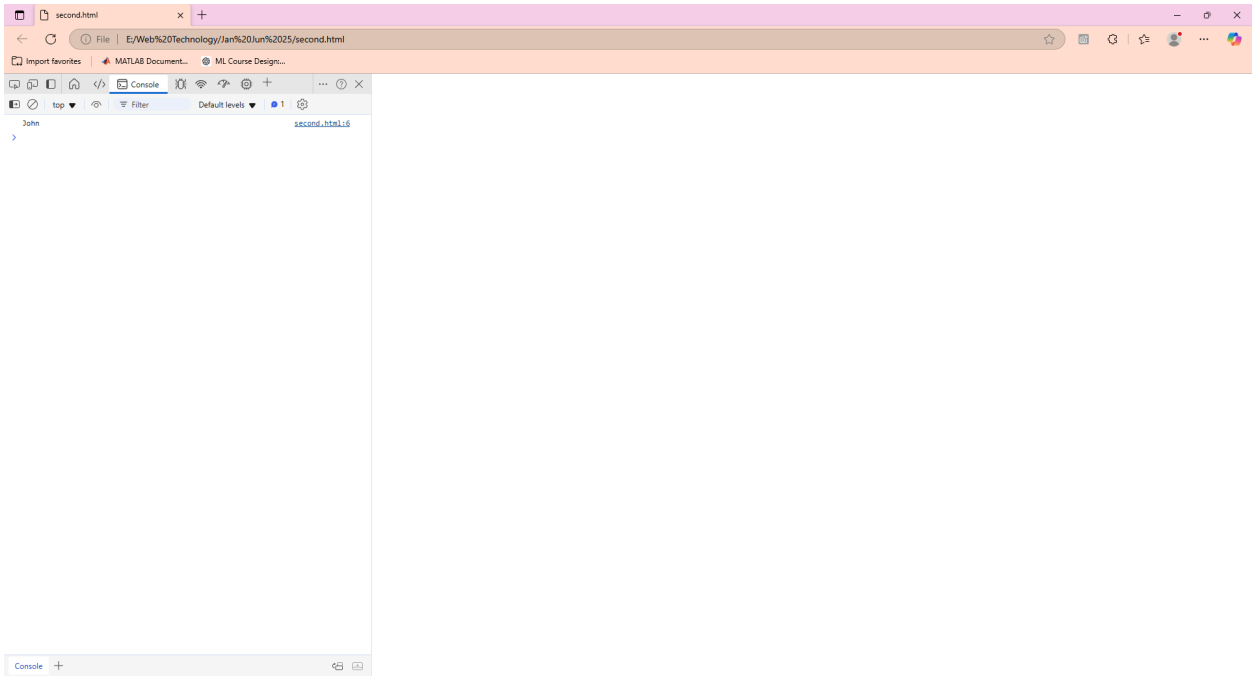
- It is easy to read and write.
- It is used for data exchange between a server and a web application.
- JSON is the standard format for APIs.

5. Parsing JSON in JavaScript

JavaScript provides methods to convert JSON strings into objects and vice versa.

Parsing JSON (String to Object)

```
let jsonString = '{"name": "John", "age": 30, "city": "New York"}';
let user = JSON.parse(jsonString);
console.log(user.name); // Output: John
```



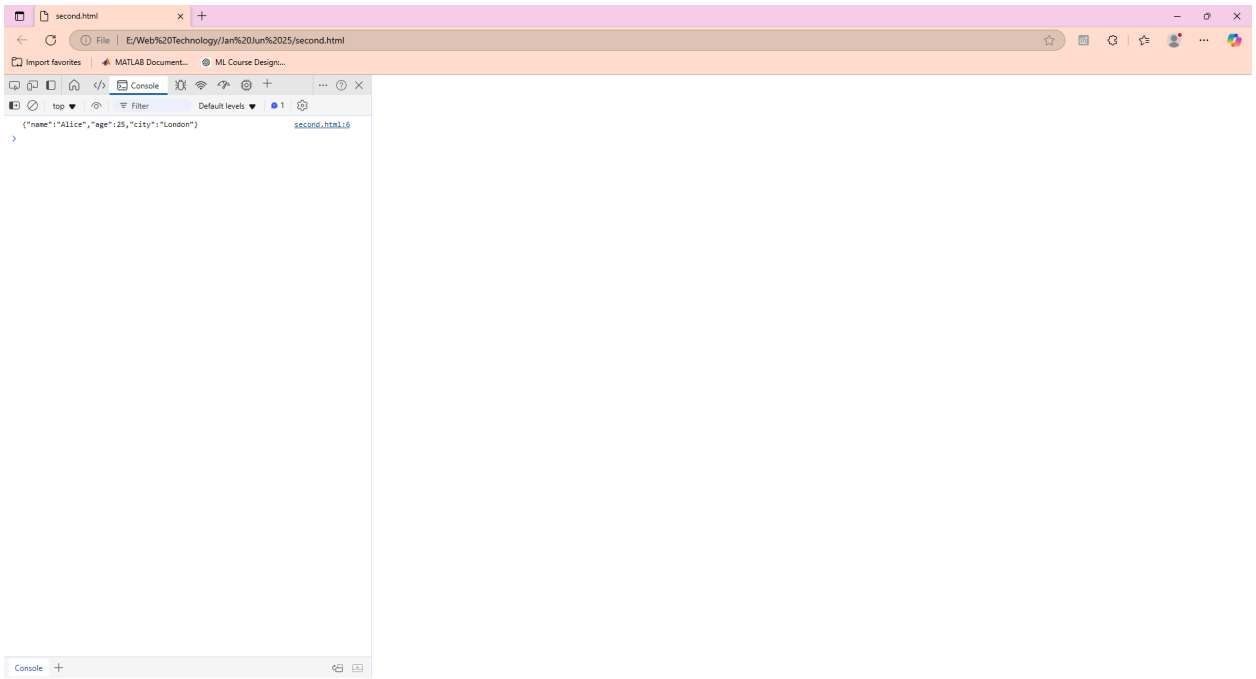
- JSON.parse() converts a JSON string into a JavaScript object.

Stringifying JavaScript Objects (Object to String)

let person = { name: "Alice", age: 25, city: "London" };

let jsonPerson = JSON.stringify(person);

console.log(jsonPerson);



- JSON.stringify() converts a JavaScript object into a JSON string.

6. Handling Errors in Fetch API

Error handling is crucial when working with network requests.

Example of Handling HTTP Errors

```
fetch('https://jsonplaceholder.typicode.com/invalid-url')
 .then(response => {
  if (!response.ok) {
   throw new Error(`HTTP error! Status: ${response.status}`);
  }
  return response.json();
 })
 .then(data => console.log(data))
 .catch(error => console.error('Error:', error));
```

- The response.ok property checks if the response was successful.
- If not, an error is thrown and caught in the .catch() block.

7. Using Async/Await with Fetch API

The Fetch API can also be used with async/await for cleaner and more readable code.

Example: Fetching Data with Async/Await

```
async function fetchData() {
 try {
  let response = await fetch('https://jsonplaceholder.typicode.com/posts/1');
  if (!response.ok) {
   throw new Error(`HTTP error! Status: ${response.status}`);
  }
  let data = await response.json();
  console.log(data);
 } catch (error) {
  console.error('Error:', error);
 }
}
```

```
fetchData();
```

Advantages of Async/Await

- Avoids callback nesting (.then() chaining).
- Improves readability and maintainability.
