MUTATION OBSERVER

ARAVIND G

What is Mutation Observer?

A MutationObserver is a built-in JavaScript object that allows you to **listen for changes** to the **DOM (Document Object Model)** tree.

It provides a way to asynchronously observe changes in the structure of a DOM tree, including **additions, removals, and modifications of elements** or attributes.

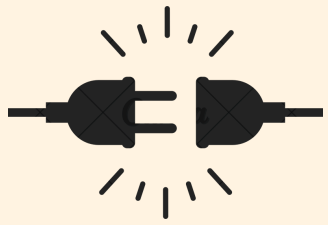
Key Concepts

MutationObserver Constructor: This creates a new MutationObserver instance.

observe() Method: This method is used to configure and start the observation of a target DOM node.

disconnect() Method: This stops the MutationObserver from observing any changes.

.



HTML

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-
scale=1.0">
    <title>MutationObserver Example</title>
</head>
<body>
    <div id="observed-div" data-info="initial">Initial content</div>
    <button id="modify-button">Modify Content</button>
    <button id="change-attribute-button">Change Attribute</button>

    <script src="script.js"></script>
</body>
</html>
```

Javascript

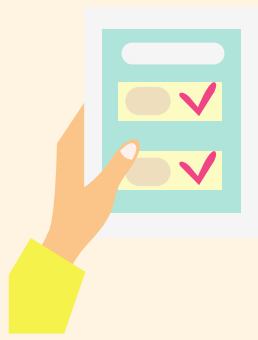
```javascript
// Select the target node
const targetNode = document.getElementById('observed-div');

// Define a callback function to execute when mutations are observed
const callback = function(mutationsList, observer) {
    for (let mutation of mutationsList) {
        if (mutation.type === 'childList') {
            console.log('Child nodes have been added or removed.');
        }
        if (mutation.type === 'characterData') {
            console.log('Character data has changed.');
        }
        if (mutation.type === 'attributes') {
            console.log(`The ${mutation.attributeName} attribute was modified from "${mutation.oldValue}"
to "${targetNode.getAttribute(mutation.attributeName)}".`);
        }
    }
};

// Create an observer instance linked to the callback function
const observer = new MutationObserver(callback);

// Start observing the target node for configured mutations
observer.observe(targetNode, {
    childList: true,              // Observes additions/removals of child nodes
    attributes: true,             // Observes changes to attributes
    characterData: true,          // Observes changes to text content
    subtree: true,                // Observes the target node and its descendants
    attributeOldValue: true,      // Records the previous value of attributes
    characterDataOldValue: true, // Records the previous value of text content
    attributeFilter: ['data-info', 'id'] // Only observes changes to specified attributes
});

// Example code to modify the content of the observed div
document.getElementById('modify-button').addEventListener('click', () => {
    targetNode.textContent = 'Content has been modified!';
});

// Example code to change an attribute of the observed div
document.getElementById('change-attribute-button').addEventListener('click', () => {
    targetNode.setAttribute('data-info', 'modified');
});
```

Attribute	Type	Description
childList	Boolean	Observes additions or removals of child nodes.
attributes	Boolean	Observes changes to the attributes of the target node.
characterData	Boolean	Observes changes to the textContent or nodeValue of the target node.
subtree	Boolean	Extends the observation to the entire subtree of the target node, including all its descendants.
attributeOldValue	Boolean	Records the previous value of any attribute that changes. Requires attributes to be true.
characterDataOldValue	Boolean	Records the previous value of textContent or nodeValue when it changes. Requires characterData to be true.

Practical Use Cases

Tracking Dynamic Content: Useful for applications that dynamically load content, such as infinite scroll implementations or single-page applications (SPAs).

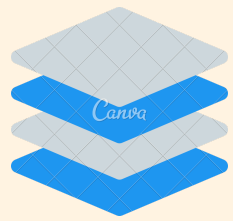
Form Validation: Monitor form inputs for changes and validate them in real-time.



UI Updates: Observe changes to the DOM and trigger UI updates or animations accordingly.

Practical Use Cases

Frameworks and Libraries: Useful in custom libraries and frameworks that need to react to DOM changes (e.g., virtual DOM implementations).



Analytics and User Behavior: Track changes to the DOM to understand user interactions and behaviors for analytics purposes.

Accessibility: Monitor and manage ARIA attributes to improve accessibility dynamically.

FRONT END DEV

Aravind G