TOPICS:

EVENT BUBBLING:--

When an event occurs, it starts with the element that triggered the event and then bubbles up to its parent elements, moving up the DOM tree. In bubbling, the event is first captured and handled by the innermost element before it moves outward to the outer elements.

Syntax:

addEventListener (type, listener, useCapture)

* Type: Use to refer to the type of event.
* Listener: Function we want to call when the event of the specified type occurs.
* User Capture: Boolean value. The Boolean value indicates the event phase. By Default use Capture is false. It means it is in the bubbling phase.

Example:--

<!DOCTYPE html>

<html>

  <head>

    <meta charset="utf-8" />

    <meta name="viewport" content="width=device-width" />

    <title>Event Bubbling</title>

  </head>

  <body>

    <div id="p1">

      <button id="c1">I am child button</button>

    </div>

    <script>

      var parent = document.querySelector("#p1");

      parent.addEventListener("click", function () {

        console.log("Parent is invoked");

      },true);

      var child = document.querySelector("#c1");

      child.addEventListener("click", function () {

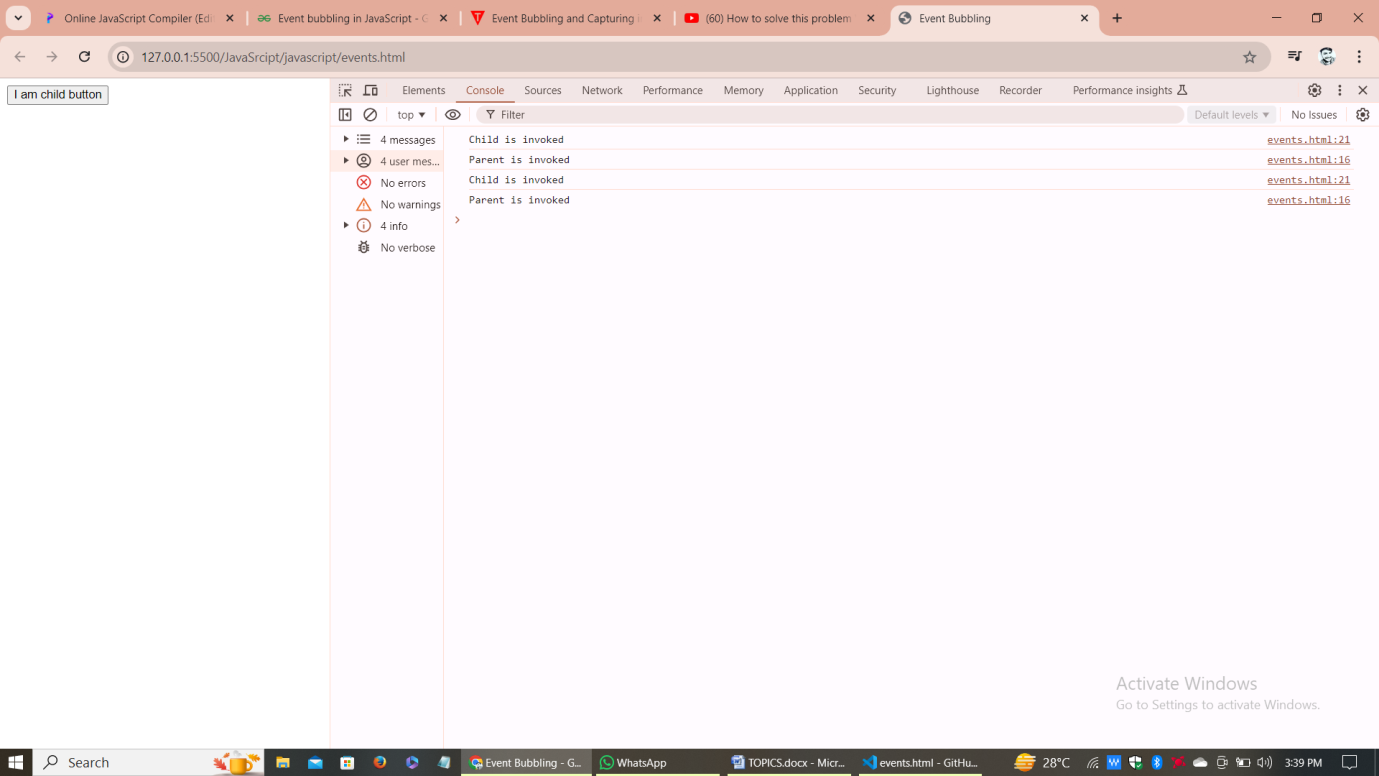
        console.log("Child is invoked");

      },true);

    </script>

  </body>

</html>

Output:-- 

EVENT CAPTURING:--

Event Capturing is opposite to event bubbling, where in event capturing, an event moves from the outermost element to the target. Otherwise, in case of event bubbling, the event movement begins from the target to the outermost element in the file. Event Capturing is performed before event bubbling but capturing is used very rarely because event bubbling is sufficient to handle the event flow.

Syntax:

selectedElement.addEventListener('event', callback, true);

Example:--

<!DOCTYPE html>

<html>

<head>

  <meta charset="utf-8">

  <meta name="viewport" content="width=device-width">

  <title>Event Capturing</title>

</head>

<body>

  <div id="p1">

    <button id="c1">I am Child</button>

  </div>

  <script>

    var parent = document.querySelector('#p1');

    var child = document.querySelector('#c1');

    parent.addEventListener('click', function(){

      console.log("Parent is invoked");

    },true);

    child.addEventListener('click', function(){

      console.log("Child is invoked");

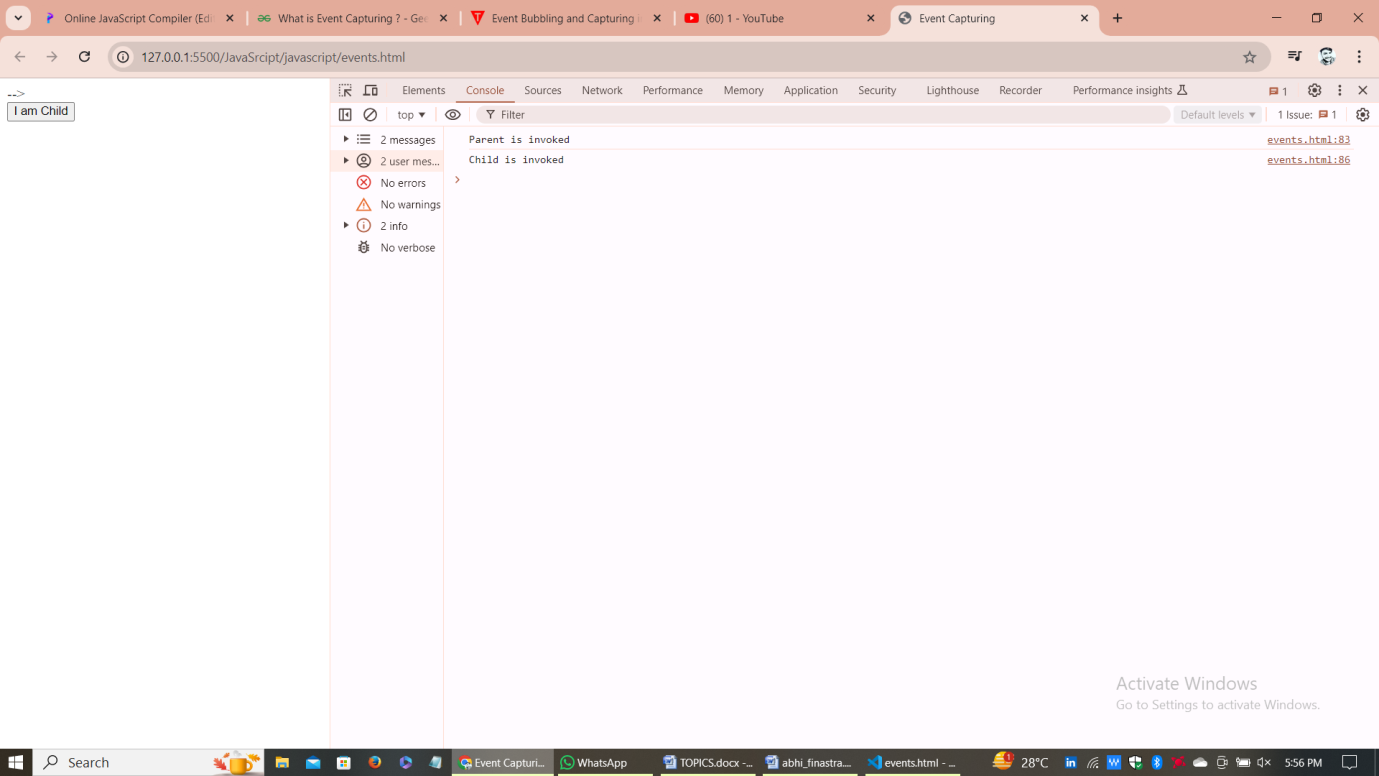
    });

  </script>

</body>

</html>

Output:--



EVENT PROPOGATION:--

In JavaScript, event propagation is a technique that controls how events move through the Document Object Model (DOM) when they happen. Ultimately, they reach the destination element, where they may trigger additional actions. It's crucial for controlling event handling in an organized way, particularly in intricate web applications with nested parts that could each have its own set of event handlers.

EVENT Delegation:--

Event Delegation is basically a pattern to handle events efficiently. Instead of adding an event listener to each and every similar element, we can add an event listener to a parent element and call an event on a particular target using the .target property of the event object.

Example:--

<!DOCTYPE html>

<html>

  <head>

    <meta charset="utf-8" />

    <meta name="viewport" content="width=device-width" />

    <title>Selecting or deselecting all CheckBoxes</title>

  </head>

  <body>

    <div>

      <ul id="category">

        <li id="Grocery">Grocery</li>

        <li id="Electronics">Electronics</li>

        <li id="Dress">Dress</li>

      </ul>

    </div>

    <script>

      document.querySelector("#category").addEventListener("click", (e) => {

        console.log(e.target.id);

      });

    </script>

  </body>

</html>

[Pure Functions](https://www.geeksforgeeks.org/pure-functions/):--

This function always returns the same output as given the same input parameters. Pure functions only depend on their input parameters and don’t affect the state of the application or other parts of the code.

The Side effect/leaking does happen when the function tries to utilize any external code inside the function, which in turn, impacts the ability of the function to perform that specific task for which the function is built.

Example:--

<!DOCTYPE html>

<html lang="en">

  <head>

    <meta charset="UTF-8" />

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

    <title>Document</title>

  </head>

  <body>

    <script>

      function geek(value) {

        return value + 100;

      }

      console.log(geek(34));

      console.log(geek(4));

      console.log(geek(12));

    </script>

  </body>

</html>

<!DOCTYPE html>

<html lang="en">

  <head>

    <meta charset="UTF-8" />

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

    <title>Document</title>

  </head>

  <body>

    <script>

      function capitalize(str) {

  return str.toUpperCase();

}

console.log(capitalize('geeks')); // Output: GEEKS

console.log(capitalize('world')); // Output: WORLD

    </script>

  </body>

</html>

Impure Functions:--

Impure functions are functions that can modify the state of the application or have side effects. In other words, impure functions can have unpredictable behavior and do affect other parts of the application.

Example:--

<!DOCTYPE html>

<html lang="en">

  <head>

    <meta charset="UTF-8" />

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

    <title>Document</title>

  </head>

  <body>

    <script>

  let flag = 0;

function incrementflag() {

    flag++;

}

incrementflag();

console.log(flag);

incrementflag();

console.log(flag);

    </script>

  </body>

</html>