DOM Structure

Document Object Model (DOM)

The Document Object Model (DOM) is a programming API for HTML and XML documents. It enables JavaScript to access and manipulate the elements and styles of a website. The browser creates a tree-like hierarchical representation of the HTML document, that tree-like structure is known as **DOM Structure** or a **DOM tree**.

Each HTML element in the DOM tree is an object. The positions of the elements in the DOM tree are nodes. The tags are element nodes. Attributes in the elements are attribute nodes. The text inside elements is a text node. It may not have children and is always a leaf of the tree. The root of the DOM tree is a `<html>` element, which is known as a **document object**.

Example:

Below, we have a simple HTML Document:

```
<!DOCTYPE HTML>

<html>

    <head>

      <title>Title goes here</title>

    </head>

    <body>

          <p> DOM Structure </p>

    </body>

</html>
```

The DOM structure for the above HTML document looks like:

```
HTML (root)

|

|---HEAD

|     |

|     |----TITLE

|            |

|            |-----#text - "Title goes here"

|

|---BODY

     |
     |----P
          |
          |----- #text - "DOM Structure"
```

Selecting elements from the DOM

JavaScript is used to get or modify the content or value of the HTML elements on the page. To perform any action on the HTML element, we need to select the target HTML element.

The ways for selecting the elements on a page are listed below:

Selecting Elements by ID

The `getElementById()` method is used to select an element based on its unique ID. The `getElementById()` method will return the element as an object if the matching element was found, or null if no matching element was found in the document.

Example: In the example below, `getElementById()`is a method, while `innerHTML` is a property. It selects the element with the `id = "demo"` attribute and changes the content to "Paragraph Changed".

```
<body>
    <p id="demo">This is a paragraph.</p>

    <script>
        document.getElementById("demo").innerHTML = "Paragraph Changed";
    </script>
</body>
```

Selecting Elements by Class Name

The `getElementsByClassName()` method used to select all the elements having specific class names. This method returns an array-like object of all child elements which have all of the given class names.

Example: selects the list of elements which have `class="test"` attribute and changes the background color as yellow.

```
<body>
<div class="test"> This is a div element with class="test". </div>
```

```
<p>
        <h1 class= "test"> This is a h1 element with class="test". </h1>
        This is a paragraph.
</p>

<p class="test">This is a p element with class="test".</p>

<script>
        var matches = document.getElementsByClassName("test");
        for(var elem in matches) {
        matches[elem].style.background = "yellow";
    }
</script>
</body>
```

Selecting Elements by Tag Name

The `getElementsByTagName()` method used to select HTML elements by tag name. This method also returns an array-like object of all child elements with the given tag name.

Example: selects all the `<p>` element and changes the background color as red.

```
<body>
    <h1> Heading </h1>
    <p>This is a paragraph of text.</p>
    <div>This is another paragraph of text.</div>
    <p>This is one more paragraph of text.</p>

    <script>
        var matches = document.getElementsByTagName("p");

        for(var elem in matches) {
        matches[elem].style.background = "red";
        }
    </script>
</body>
```

Selecting Elements with CSS Selectors

We can use `querySelector()` and `querySelectorAll()` methods to select elements that matches the specified CSS selector. The `querySelector()` finds the first element that matches a CSS selector whereas the `querySelectorAll()` finds all elements that match a CSS selector.

Example:

```
<body>
    <p id = "para">This is a paragraph</p>
    <ul>
        <li>Apple</li>
        <li>Orange</li>
        <li>Mango</li>
    </ul>

    <script>
    var matches = document.querySelectorAll("ul li");

    for(var elem of matches) {
        document.write(elem.innerHTML + "<br>"); //outputs: "Apple Orange Mango"
    }
      document.write(document.querySelector('#para').textContent); //outputs: "This i
s a paragraph"

    </script>
</body>
```

DOM Manipulation

We can add, remove, replace, and copy any element into a DOM tree. DOM Manipulation methods are listed below:

Create Elements

The `createElement()` method is used to create a new element and attach it to the DOM tree.

Example:

```
var elmt = document.createElement('div');

elmt.innerHTML = '<p>Hello World!</p>';
```

Replace Child Elements

The `replaceChild()` method is used to remove an element from the DOM tree and insert a new one in its place.

Example:

```
<body>

<div>

        This is a div element.

<div>


<script>

    // selecting the <div> element

        var elmt = document.querySelector('div');


    //creating new <p> element and adding content inside it.

        var newElmt = document.createElement('p');

        newElmt.innerHTML = '<b>The div element is replaced with p element</b>';


    //replacing the <div> element with the <p> element

        elmt.parentNode.replaceChild(newElmt, elmt);


</script>

</body>
```

Remove Child Elements

The `removeChild()` method is used to remove an element from the DOM tree.

Example: Here, first we select the element to remove, then walk up the tree to its parent and remove the child element from there.

```
var elmt = document.querySelector('div'); // select the first returned <div> element

elmt.parentNode.removeChild(elmt);
```

Append a Node

The `appendChild()` method is used to add a node to the end of the list of child nodes of a specified parent node.

Example: Here, we add three list items to the `<ul>` element

```
<body>
    <ul id="animals">
    </ul>

    <script>
        function createAnimalList(name) {
            let li = document.createElement('li');
            li.textContent = name;
            return li;
        }
        // get the ul #animals
        const list = document.querySelector('#animals');
        // add animals to the list
        list.appendChild(createAnimalList('Lion'));
        list.appendChild(createAnimalList('Tiger'));
        list.appendChild(createAnimalList('Wolf'));
    </script>
</body>
```

Insert a Node before another

The `insertBefore()` method is used to insert a node before another node as a child node of a parent node.

Example: Here, we insert the new `<li>` element before the first child of `<ul>` element.

```
<body>
    <ul id="animal">
    <li>Lion</li>
    <li>Tigerr</li>
    </ul>
```

```
   <script>
        let animal = document.getElementById('animal');

        // create a new li node
        let li = document.createElement('li');
        li.textContent = 'Wolf';

        // insert a new node before the first list item
        animal.insertBefore(li, animal.firstElementChild);
    </script>
</body>
```

Insert a Node after another

JavaScript DOM provides the `insertBefore()` method that allows you to insert a new after an existing node as a child node. However, it hasn't supported the insertAfter() method yet.

So, we can insert a new node after an existing node as a child node, by selecting a parent node of the existing node and call the `insertBefore()` method on the parent node to insert a new node before that immediate sibling node.

Example: Here, we inserts the new `<li>` element after the first child of `<ul>` element.

```
<body>
    <ul id="animal">
    <li>Lion</li>
    <li>Tiger</li>
    </ul>

   <script>
        let animal = document.getElementById('animal');

        // create a new li node
        let li = document.createElement('li');
        li.textContent = 'Wolf';

        // insert a new node before the first list item
        animal.insertBefore(li, animal.firstElementChild.nextSibling);
    </script>
</body>
```

Get or Set text of a Node

The `textContent` property is used to get and set a text content inside a particular node.

Example:

```
<body>
    <div id = "content">
        This is div element.
    </div>
   <script>
                 // Getting a text content
        let content = document.getElementById('content');
                 alert("Getting a text content inside div element: " +  content.textC
ontent);

        //setting a text content
       content.textContent = 'New text content in the div element';

    </script>
</body>
```

Get or Set HTML of Element

The `innerHTML` property to get the text and inner HTML elements inside a particular element and the setting will replace the existing text and inner tags inside an element with the new content.

innerHTML vs. textContent - The `innerHTML` property returns the text and inner HTML elements. The `textContent` property returns only the text Content.

Example:

```
<body>
  <div id="myBdy">
    <p id = "para">This is Paragraph.</p>

    <button onclick="myFunction()"  >Try it</button>
    <p id="demo"></p>


  </div>
<script>
  function myFunction() {
      // get HTML of Element
    var x = document.getElementById("para").innerHTML;
    document.getElementById("demo").innerHTML = x;
  }
// You can understand the difference between innerHTML and textContent property clear
ly from the output of the
// below two alert boxes
  alert ("textcontent property:" + document.getElementById("myBdy").textContent);
  alert ("innerHTML property:" + document.getElementById("myBdy").innerHTML);
</script>


</body>
```

Clone a Node

The `cloneNode()` method is used to clone an element. The cloneNode() method accepts an optional parameter. If the parameter value is `true`, then the original node and all of its descendants are cloned. If the parameter value is false, only the original node will be cloned.

Example: Here, the parameter value for `cloneNode()` method is `true`. So it clones the target node and all of its descendants.

```
<body>
    <ul id="animal">
        <li>Lion</li>
        <li>Tiger</li>
        <li>Wolf</li>
    </ul>
   <script>
        let list = document.querySelector('#animal');
        let clonedList = list.cloneNode(true);
        clonedList.id = 'cloned animal';
        document.body.appendChild(clonedList);
    </script>
</body>
```

Managing Attributes

- `getAttribute(attribute_name)` method - Used to get the value of an attribute on a specified element
- `setAttribute(attribute_name, attribute_value)` method - Used to set a value of an attribute on a specified element,
- `removeAttribute(attribute_name)` method - Used to remove an attribute with a specified name from an element
- `hasAttribute(attribute_name)` method - Used to check an element has a specified attribute or not.

Traversing the DOM

The DOM allows us to do anything with elements and their contents, but first we need to reach the corresponding DOM object. The properties used to transverse the DOM or to reach/get a particular DOM object are listed below:

Topmost nodes

The `document` object is the root of every node in the DOM. The topmost tree nodes are associated with document properties:

- `document.documentElement` property - refers the DOM node of the `<html>` tag.
- `document.head` property - refers the DOM node of the `<head>` tag.
- `document.body` property - refers the DOM node of the `<body>` tag.

Parent Nodes

The parent of any node is the node that is one level above in the DOM tree. There are two properties to get the parent — `parentNode` and `parentElement`. The `parentNode` is most commonly used for traversing the DOM.

Example:

```
<body>
    <div id="main">
```

```
        <p id="para">This is a note!</p>
    </div>


    <script>
        let elmt = document.querySelector('#para');
        document.write(elmt.parentNode+ "<br>"); // outputs: [object HTMLDivElement]
    </script>
</body>
```

The `parentElement` property returns the “element” parent, whereas `parentNode` returns “any node” parent. With the one exception of `document.documentElement`:

```
alert( document.documentElement.parentNode ); // document
alert( document.documentElement.parentElement ); // null
```

Child Nodes

The children of a node are the nodes that are one level below it. The properties are listed below:

- `childNodes` property - returns a list of child nodes, including text nodes.
- `firstChild` property - give access to the first child node.
- `lastChild` property - give access to the last child node.

Example:

```
<body>
        <div id="myDiv">
        <p>This is a paragraph - first child</p>
        <div> this is a div elemt - last child</div>
    </div>
    <script>
        let elmt = document.querySelector('#myDiv');
        document.write("<br> Child nodes of div element: <br>");
        for (let i = 0; i < elmt.childNodes.length; i++) {
              document.write(elmt.childNodes[i]  + "<br>");
              }


       document.write("<br> First child of div element: <br>" +elmt.firstChild) ;
       document.write("<br> Last child of div element: <br>" +elmt.lastChild) ;
    </script>
</body>
```

The output will be:

```
This is a paragraph - first child


this is a div elemt - last child


Child nodes of div element:
[object Text]
[object HTMLParagraphElement]
[object Text]
[object HTMLDivElement]
[object Text]


First child of div element:
[object Text]
Last child of div element:
[object Text]
```

Sibling Nodes

Siblings are nodes that are children of the same parent. The siblings of a node are any node on the same tree level in the DOM.

- `previousElementSibling` property - give access to the previous sibling Node, i.e. the node that immediately precedes the specified node.
- `nextElementSibling` property - give access to the next sibling node, i.e. the node that immediately precedes or follows the specified node.

Example:

```
<body>
    <ul >
        <li>list item 1</li>
        <li class="list">list item 2</li>
        <li>list item 3</li>

    </ul>

    <script>

       const secondListItem = document.querySelector('.list');

        document.write(secondListItem.previousElementSibling.textContent) ;  // outpu
ts:  "list item 1"

       document.write(secondListItem.nextElementSibling.textContent);   //outputs: "l
ist item 3"

    </script>

</body>
```

Events & Listeners

JavaScript Events

Events occur when user interaction takes place on a web page, such as when the end-user clicked a link or button, pressed a key on the keyboard, moved the mouse pointer, submits a form, etc. The browser also triggers the events, such as the page load and unload events.

When an event occurs, we use a JavaScript event handler (or an event listener) to detect them and perform a specific task.

Some of the commonly used JavaScript Events are listed below:

- **onclick** - This is a click event occurs when a user clicks the element on a web page.
- **ondblclick** - This is a click event occurs when a user double clicks the element on a web page.
- **onload** - This is a load event occurs when the browser has finished loading the page.
- **onunload** - This is a load event occurs when a user closes the document.
- **onresize** - This is a load event occurs when the browser window is minimized or maximized.
- **onmouseover** - This is a mouse event occurs when the user moves the mouse over an HTML element.
- **onmouseout** - This is a mouse event occurs when the user moves the mouse away from an HTML element.
- **onkeydown** - This is a keyboard event occurs when the user presses down a key on the keyboard.
- **onkeyup** - This is a keyboard event occurs when the user releases a key on the keyboard.

- **onfocus** - This is a form-input event occurs when the user gives focus to an element on a web page.
- **onblur** - This is a form-input event occurs when the user takes the focus away from a form element or a window.
- **onchange** - This is a form-input event that occurs when a user changes the value of a form element.
- **onsubmit** - This is a form-input event that occurs when the user submits a form on a web page.

Example:

```
<body>
    <button onclick= onclickEvent()>Click me!!</button>
    <p id= "para" onmouseover = onmouseoverEvent() onmouseout = onmouseoutEvent() >This is a Paragraph....</p>
    <script>
        function onclickEvent(){
            alert('Hello, You clicked the button');
        }
        function onmouseoverEvent(){
            document.getElementById("para").style.backgroundColor = "green";
        }
        function onmouseoutEvent(){
            document.getElementById("para").style.backgroundColor = "yellow";
        }
    </script>
</body>
```

JavaScript EventListener

An event listener is a function in JavaScript that waits for an event to occur.
The `addEventListener()` function is an inbuilt function in JavaScript used to handle the event.

The Syntax of addEventListener() function: `element.addEventListener(event, function, useCapture)`

Where,

- event - Specifies the name of the event.
- function - Specifies the function to run when the event occurs

- useCapture - It is an optional parameter takes a boolean value. If the parameter value is true then the event handler is executed in the capturing phase. If the parameter value is false then the event handler is executed in the bubbling phase.

The `removeEventListener()` method used to remove an event handler that has been attached with the `addEventListener() method`.

Example:

```
<body>
    <h2>JavaScript addEventListener()</h2>
    <p id="myBtn">This is a paragraph.</p>
    <p id="demo"></p>
    <script>
        var x = document.getElementById("myBtn");
        x.addEventListener("mouseover", myFunction);
        x.addEventListener("mouseout", mySecondFunction);

        function myFunction() {
        document.getElementById("demo").innerHTML += "Moused over!<br>";
        x.style.backgroundColor = "green";
        }
        function mySecondFunction() {
        document.getElementById("demo").innerHTML += "Moused out!<br>";
        x.style.backgroundColor = "white";
        }

    </script>
</body>
```

Bubbling, capturing

Event Bubbling

In Event Bubbling, the event propagates from the target element to its parents, then all its ancestors that are on the way to top. Bubbling follows the **Bottom to Top** approach.

Example: Event Bubbling works for all handlers, regardless of how they registered with the `addEventListener()`. When we click on any element, event propagates or bubbles back up the DOM tree, from the target element up to the Window, visiting all of the ancestors of the target element one by one. (a-> p -> div)

```
<body>
<div onclick="alert('Bubbling: ' + this.tagName)">DIV
    <p onclick="alert('Bubbling: ' + this.tagName)">P
        <a href="#" onclick="alert('Bubbling: ' + this.tagName)">Click Me!!</a>
    </p>
</div>
</body>
```

If we click on the `<a>` element in the above example, it results in the alert pop-ups in below order:

1. alert pop-ups saying 'Bubbling: a'
2. alert pop-ups saying 'Bubbling: p'
3. alert pop-ups saying 'Bubbling: div'

Event Capturing

In Event Capturing, the event propagates from the top element to the target element. Capturing follows the **Top to Bottom** approach.

Example: Event capturing only works with event handlers registered with the `addEventListener()` method when the third argument is set to true. When we click on the any element, the event capturing propagates the element from top element to target element (div -> p -> a).

```
<body>
<div id="wrap">DIV
    <p class="hint">P<br>
        <a href="#">Click Me!!</a>
    </p>
</div>


<script>
    function showTagName() {
        alert("Capturing: "+ this.tagName);
    }


    var elems = document.querySelectorAll("div, p, a");
    for(let elem of elems) {
        elem.addEventListener("click", showTagName, true);
    }
</script>
```

If we click on the `<a>` element in the above example, it results in the alert pop-ups in below order:

1. alert pop-ups saying 'Capturing: div'
2. alert pop-ups saying 'Capturing: p'
3. alert pop-ups saying 'Capturing: a'

Event Target

Event Target is the target element that has generated the event in DOM.
The `event.target` is used to access the target element.

Stopping the Event Propagation

- event.stopPropagation() method

It used to stop the event to travel to the bottom to top i.e. Event Bubbling. If you want to stop the event flow from event target to top element in DOM, we use `event.stopPropagation()` method.

- event.stopImmediatePropagation() method

If an element has multiple event handlers on a single event, then even if one of them stops the bubbling, the other ones still execute. The `event.stopPropagation()` stops event bubbling but all other handlers will run. To stop the bubbling and prevent handlers on the current element from running, we use `event.stopImmediatePropagation()` method.

let and const keywords

In ES6, `const` and `let` keywords allow developers to declare variables in the block scope, which means those variables exist only within the corresponding block.

Variables declared with the `let` and `const` keyword can have **Block Scope**.

`let` Keyword

The variables declared inside a block {} have Block Scope, so it cannot be accessed from outside the block.

Example:

```
{
  let y = 2;
}
// y can NOT be used here

var x = 10;
// Here x is 10
{
  let x = 2;
  // Here x is 2
}
// Here x is 10
```

`const` Keyword

`const` is block-scoped, much like variables defined using the `let` keyword. The value of a `const` variable can't be changed through reassignment, and it can't be re-declared.

Example: We cannot change the primitive value of constant variable.

```
const a = 12;
a = a+11; //error
a="hello"; //error
```

Example: We can change the properties of a constant object but we can't reassign the constant object.

```
//const object
const person = { name: "Johnson" , age: "23", gender: "male"};

// change a property
person.age = "21";

console.log(person); // prints " {name: "Johnson", age: "21", gender: "male"} " in th
e console.

//reassigning const object
```

```
person = { name: "Mercy" , age: "23", gender: "female"}; //error
```

ES6 Template Literals

Template literals are a new feature introduced in ECMAScript 2015/ ES6. It provides an easy way to create multiline strings and perform string interpolation. Template literals are the string literals and allow embedded expressions.

Template literals are strings that enclosed within the backtick character`(`)`. Template literals allow for embedded expressions (placeholders), which are indicated by the dollar sign and curly braces (`$(expression}`). These expressions can be used to evaluate code.

Syntax: ``var str = `string value` ``

Multiline strings

We use the escape character, represented as `\n`, to give a new line for creating a multiline string. In Template Literals, there is no need to add `\n` at the end when the string enclosed within the backtick (`` ` ``) character. Template literals used to create the new line by literally moving a string to a new line without using any character.

Example:

```
console.log(`I am the first line
     I am the second line
      I am the third line`);
```

Output:

```
I am the first line
I am the second line
I am the third line
```

String Interpolation

In JavaScript, the template literals give support for string interpolation. Template literals perform string interpolation using embedded expressions, `${}`, which are replaced with the value of the code within it.

Example:

```
function sayHello(){
    return "Hello World!!!"
}
```

```
var x = 10;
var y = 20;
document.write(`${sayHello()}, The product of the variables ${x} and ${y} are
 ${x*y}`);
```

Output on the console:

```
Hello World!!!, The product of the variables 10 and 20 are 200.
```

Tagged templates:

Tagged templates are one feature provided by ES6. Tagged templates allow us to parse template literals with a function.

The following is a tagged template:

`tagFunction`Hello ${firstName} ${lastName}!``

Putting a template literal after an expression triggers a function call, similar to how a parameter list triggers a function call.

The above code is equivalent to the following function call:

`tagFunction(['Hello ', ' ', '!'], firstName, lastName)`

Thus, the name before the content in backticks is the name of a function to call, the tag function. The first argument of the tag function contains an array having string values, like `['Hello ', ' ', '!']` in the above tag Function. The remaining arguments are substitutions delimited by `${}`, such as `firstName` and `lastName` in the above tag Function.

Example:

```
function printAll(literalArray, operator1, operator2, result){
 console.log(literalArray);
 console.log(operator1);
 console.log(operator2);
 console.log(result);
}
a = 3;
b = 4;
printAll `Addition:  ${a} + ${b} = ${a+b}`;
```

Output on the console:

```
(4) ["Addition:  ", " + ", " = ", "", raw: Array(4)]
3
```

```
4
7
```

Raw Strings

The template literal raw method allows the accessing of raw strings as they were entered.

The `String.raw()` is a built-in function which accepts a string literal argument and returns raw string. It returns the strings without any interpretation of backslashed characters.

Example:

```
var a =3;
var b =4;
var myString = String.raw`sum:\n ${a+b}`;
console.log(myString);
```

Output on the console:

```
sum:\n 7
```

The `raw` is a special built-in property which can be used on the first function argument of tagged templates. This allows us to access the raw strings as they were entered.

Example:

```
function indexTest(literalArray){
 console.log(literalArray.raw[0]);
 console.log(literalArray[0]);
}
indexTest `"finding \n errors"`;
```

Output on the console:

```
"finding \n errors"
"finding
 errors"
```

Arrow Functions

Arrow functions, introduced in ES6, provides a concise way to write functions in JavaScript. They save developers time and simplify function scope.

- **One-line arrow functions** have implicit return and defined without curly braces.

```
let func = (arg1, arg2, ...argN) => expression;
```

Example:

```
var hello = () => "Hello World!";
hello(); // output: "Hello World!"
```

- **Multiline arrow functions** have multiple statements inside the function, enclosed with curly braces and need an explicit return to return something.

```
let func = (arg1, arg2, ...argN) =>{
stament1;
...
statementn;
return value;
}
```

Example:

```
var sum = (a, b) => {
      let result =  a + b;
    return "Sum is" + result;
};

sum(5, 3);  //output: "sum is 8"
```

Promises

Promises are used to handle asynchronous operations in JavaScript.

The constructor syntax for a promise object is:

```
let promise = new Promise(function(resolve, reject) {
  // executor (the producing code, "singer")
});
```

The function passed to `new Promise` is called the **executor**. When a `promise` object is created, the executor runs automatically. It contains the code which produces the result. The arguments `resolve` and `reject` are callbacks.

When the executor obtains the result, it should call one of these callbacks:

- `resolve(value)` — if the job finished successfully, with the result `value`.
- `reject(error)` — if an error occurred, the error is the `error` object.

The `Promise.status` property, gives information about the state of the `Promise` object. The promise object can have three states: **pending**, **fulfilled**, and **rejected**.

A Promise object connects the executor and the consuming functions which will receive the result or error. Consuming functions can be registered using methods `.then`, `.catch` and `.finally`.

Example:

```
var promise = new Promise(function(resolve, reject) {
  const x = 5;
  const y = 3;
  if(x >= y) {
    resolve();
  } else {
    reject();
  }
});


promise.
    then(function () {
        console.log('Success! x have greater value');
    }).
    catch(function () {
        console.log('Error');
    });
```

AJAX Workflow with XmlHttpRequest Object

AJAX

AJAX stands for **A**synchronous **J**avaScript And **X**ML. Ajax describes the process of exchanging data from a web server asynchronously with the help of XML, HTML, CSS, and JavaScript. It just loads the data from the server and selectively updates some webpage parts without refreshing the page. Ajax uses the browser's built-in XMLHttpRequest (XHR) object to send and receive data to and from a web server asynchronously, in the background, without blocking the page or interfering with the user's experience.

Despite the name, modern usage of AJAX usually works with JSON data rather than XML as we will see.

Ajax Work Flow

Ajax works flow starts from the client-side, on the browser. The steps involved in Ajax communication as follows:

1. A client event occurs on a webpage. (for example, the user clicks a button)
2. JavaScript creates an XMLHttpRequest object.
3. The XMLHttpRequest object makes an asynchronous request to the server.
4. The server process the received HttpRequest.
5. The server creates a response and sends data back to the browser.
6. Browser process the returned data using JavaScript.
7. The page content updated by JavaScript.

The XMLHttpRequest Object

The keystone of AJAX is the XMLHttpRequest object. An XMLHttpRequest (XHR) object used to make HTTP requests to the server and receive data in response.

Create an XMLHttpRequest Object:

Before you perform Ajax communication between client and server, we should create an XMLHttpRequest object.

```
var xhttp = new XMLHttpRequest();
```

Send a Request To a Server:

To send a request to a server, we use the open() and send() methods of the XMLHttpRequest object.

- **open(method, URL, async)**

where, method — Specifies the HTTP request method to use, such as "GET", "POST", etc., URL - Specifies the location of the server async - Specifies whether the request should be handled asynchronously or not. If "true" then the script processing carries without waiting for a response. If "false" then the script waits for a response.

- **send()** - Sends the request to the server. It accepts an optional parameter that allows us to specify the request's body (used for POST).

In the GET method, the data is sent as URL parameters. In the POST method, the data is sent to the server as a part of the HTTP request body, which is not visible in the URL.

```
xhttp.open("GET", "ajax_info.txt", true);

xhttp.send();
```

Server Response

The Server Response returned in the form of responseText or responseXML or status or statusText.

- **responseText** - Returns the response as a string.
- **responseXML** - Returns the response as XML.
- **status** - Returns the status as a number (For example, 200: "OK", 403: "Forbidden",404: "Page not found")
- **statusText** -Returns the status as a string (e.g., "Not Found" or "OK").

We have to wait for the data to be available to process it, and in this purpose, the state of availability of data is given by the **readyState** attribute of XMLHttpRequest.
The **onreadystatechange** function is called every time the readyState changes.

The readyState property defines the status of the XMLHttpRequest:

- readyState = 0 : Not initialized
- readyState = 1 : Connection established
- readyState = 2 : request received
- readyState = 3 : processing request
- readyState = 4 : request finished and response is ready

Example:

```
<script>

function loadDoc() {

  var xhttp = new XMLHttpRequest();

  xhttp.onreadystatechange = function() {

    if (this.readyState == 4 && this.status == 200) {

      document.getElementById("demo").innerHTML =

      this.responseText;

    }
  };
  xhttp.open("GET", "ajax_info.txt", true);
  xhttp.send();
}
</script>
```

Working with JSON in Ajax

Sending JSON request payload and receiving the JSON response object are common tasks while dealing with AJAX. JavaScript retrieves the JSON data sent by the server, parse them, and displaying them on the webpage.

JSON in Request Payload

In JavaScript, to send a request using JSON data, we need to serialize our JSON object into a string. The `JSON.stringify()` method is used to converting an object to a string. Then, the server receives the string and process the request.

Example:

```
var data = {"name" : "Matt"};
var xmlhttp = new XMLHttpRequest();
xmlhttp.open("POST", "/demo", true);
//Use stringify() method to get string
xmlhttp.send( JSON.stringify( data ) );
```

JSON in Response Body

If the response from the server is string/text, we need to parse them into a JSON object. The `JSON.parse()` method converts a JSON string representation to a JSON object.

Example:

```
var xmlhttp = new XMLHttpRequest();

xmlhttp.onreadystatechange = function() {
    if (this.readyState == 4 && this.status == 200) {
        //Use parse() method to convert JSON string to JSON object
        var responseJsonObj = JSON.parse(this.responseText);

        console.log( responseJsonObj.name );
        console.log( responseJsonObj.age );
    }
};

xmlhttp.open("GET", "/demo", true);

xmlhttp.send();
```

XML in Response Body

While we normally wish to parse our response from JSON, sometimes we might be dealing with an older server and we need to parse XML data. The **`responseXML`** field contains the data as a read-only Document Object Model.

Example:

```
var xmlhttp = new XMLHttpRequest();

xmlhttp.onreadystatechange = function() {
    if (this.readyState == 4 && this.status == 200) {
        var responseXml = this.responseXML;

        console.log( responseXml.getElementsByTagName('myTag') )
    }
};

xmlhttp.open("GET", "/demo", true);

xmlhttp.send();
```

Fetch API

Instead of using XMLHttpRequest enabled Ajax, we can use Fetch API, which is modern and versatile. The Fetch API provides a `fetch()` method defined on the `window` object. This method used to send requests and returns a `Promise` that retrieved from the response.
A **Promise** object represents a value that may not be available now but, will be resolved in the future. It allows us to write asynchronous code.

The syntax for fetch() method: `let promise = fetch(url, [options])` The browser requests the server and returns a promise as a response. When the request unable to make HTTP-request due to network problems or response has failure HTTP-status code is 404 or 500, then the Fetch API rejects the Promise object. When we get a response successfully form the server, the promise object returned in the **Response Body**.

The methods to access the response body in various formats:

- `response.text()` – read the response and return as text.
- `response.json()` – parse the response as JSON.
- `response.formData()` – return the response as FormData object .
- `response.blob()` – return the response as Blob (binary data with type).
- `response.arrayBuffer()` – return the response as ArrayBuffer(low-level representation of binary data).

We also use the `async` and `await` keyword with the `fetch()` method. The `async` keyword is added to functions to tell them to return a promise rather than directly returning the value. The `await` keyword only works inside async functions, used to pause the code on that line until promise gets complete.

Example:

```
async function asyncFunc() {
  let response = await fetch(protectedUrl);
  let text = await response.text(); // response body consumed
  document.write(text);
}

asyncFunc();
```

Response headers - The response headers are available in a Map-like headers object in `response.headers`. To get individual headers by name or iterate over them.

Example:

```
async function asyncFunc() {
  let response = await fetch(githubUrl);

  // get one header
  alert(response.headers.get('Content-Type')); // application/json; charset=utf-8

  // iterate over all headers
  for (let [key, value] of response.headers) {
    alert(`${key} = ${value}`);
  }
}


asyncFunc();
```

Request headers - To set a request header inside the `fetch` method, we can use the `headers` attribute.

Example:

```
let response = fetch(protectedUrl, {
  headers: {
    Authentication: 'secret'
  }
});
```

POST Request:

To make a POST request, we need to mention the HTTP method (`method`) and request the body (`body`) inside the fetch method. The request body can be string, FormData object, or blob. If the request `body` is a string, then `Content-Type` header is set to `text/plain;charset=UTF-8`. If the request `body` is a JSON, then `Content-Type` header is set to `application/json;charset=UTF-8`. We don't set `Content-Type` manually for blob object.

Example:

```
async function asyncFunc() {
  let user = {
    name: 'John',
    surname: 'Smith'
  };
  // url is a server location
  let response = await fetch(url, {
    method: 'POST',
    headers: {
      'Content-Type': 'application/json;charset=utf-8'
    },
    body: JSON.stringify(user)
  });


  let result = await response.json();
  alert(result.message);
}

asyncFunc();
```

Handling Errors

The Fetch API generates a promise, meaning that if the request fails, it will cause the promise to enter the `reject` state. To handle this, we need to either surround our `await` instruction with a `try...catch` block or to append a `catch()` callback to our promise.

Example:

```
async function asyncFunc() {
  let user = {
    name: 'John',
    surname: 'Smith'
  };
  // url is a server location
  let response = await fetch(url, {
    method: 'POST',
    headers: {
      'Content-Type': 'application/json;charset=utf-8'
    },
    body: JSON.stringify(user)
  });
  try {
    let result = await response.json();
    alert(result.message);
  } catch (error) {
    console.error(error);
  }
}

asyncFunc();
```

Example:

```
let user = {
  name: 'John',
  surname: 'Smith'
};
// url is a server location
let response = fetch(url, {
  method: 'POST',
  headers: {
    'Content-Type': 'application/json;charset=utf-8'
  },
  body: JSON.stringify(user)
}).then((response)=>{
  let result = response.json();
  alert(result.message);
}).catch((error)=>{
  console.error(error);
});
```