Iframe

**Change the Background:**

By Using getElementById () Method we can access the <iframe> element and stored inside the variable x and by using style.backgroundColor of Javascript method we can change the Background color of the Iframe

**Create an Iframe Object**

You can create an <iframe> element by using the document.createElement() method we can set the Attribute src by using setAttribute Method in javascript and append the iframe in the body of the document

**Removing an Iframe:**

We can access the Iframe element by using getElemetById() and remove it by using remove() method in Javascript

**HiddenIframe:**

* + - * The [hidden](https://www.dofactory.com/html/hidden) attribute hides the [<iframe>](https://www.dofactory.com/html/iframe) element.
      * We can specify either 'hidden' (without value) or 'hidden="hidden"'.
      * A hidden [<iframe>](https://www.dofactory.com/html/iframe) is not visible, but maintains its position on the page.

**Minimized Iframe**:

The I frame can be Minimised by reducing the default width and height of the I frame by using set Attribute(“width”,”100px”)

**Transparent iframe**:

We can make the Iframe transparent by specifying the opacity CSS property to 0.

**Input Methods**

**Create Input:**

By using createElement() method of Java script we can dynamically create input elements text ,password ,email etc

**Hidden Input:**

<input type=” hidden”> by using the attribute type=” hidden “ we can create the Hidden Input field. We can access the value of the hidden input by using its Id

**Remove Input:**

remove () method in JavaScript can be used to delete the input filed dynamically.

document. getElementById(“id “).remove()

Get Text Value

**Methods:**

We can get the value of the input text filed by the following methods

getElementById()

getElementsByName()

getElementsByClassName()

getElementsByTagName()

document.querySelector()

Java Script

**Inline Script:**

Without using <script> Tag in Html we can create Java script file by using create Element(“script”) method of JavaScript and append the script in the body or head element of html

**External Script:**

We can create scriptload () function and send the filename as the src attribute and load the second file from the main file and load the third file from the second file

Index.html

Main.js

External file

( Second file)

Third file

**Third Party Script:**

We can load third party script like bootstrap inside our script file.

**Keyboard Event**

* The onKeydown Event Happens when the user is pressing a key
* onKeyPress Event occurs when the user presses the key
* onKeyDownevent occurs when the user Releases a key

**Keydown Global:**

After Pressing the key the window.addEventListener() calls the corresponding functions and display the type of the event and what key has typed

**MEDIA**

**GeoLocations:**

The HTML Geolocation API is used to get the geographical position of a user.

Since this can compromise privacy, the position is not available unless the user approves it.

If (navigator. geolocation) checks whether our browser supports geolocations If supported, run the getCurrentPosition() method. If not, display a message to the user

* If the getCurrentPosition() method is successful, it returns a coordinates object to the function specified in the parameter (showPosition)
* The showPosition() function outputs the Latitude and Longitude

We can pass the fetchapiurl in the fetch method to get the countryname state and city(data.counrty)

**WebCamera & MicroPhone:**

With getUserMedia(), we can access webcam and microphone input without a plugin.

navigator. mediaDevices.getUserMedia:

To use the webcam or microphone, we need to request permission. The parameter to getUserMedia () is an object specifying the details and requirements for each type of media you want to access. For example, if we want to access the webcam, the parameter should be {video: true}. To use both the microphone and camera, pass {video: true, audio: true}:

 It works in conjunction with our other HTML5 elements <audio> and <video>. don’t set a src attribute or include <source> elements on the <video> element. Instead of the URL of a media file, we give the video a Media Stream from the webcam.

Using the document. querySelector we can access the video tag of HTML Element set the srcattribute as media stream and video. Play () will start the camera and microphone

**Workers**

**Web Workers**

A web worker is a JavaScript that runs in the background, independently of other scripts, without affecting the performance of the page.

This example creates a simple web worker that count numbers in the background:

Before creating a web worker, check whether the user's browser supports it:

if (typeof(Worker) !== "undefined") {  
  // Yes! Web worker support!  
  // *Some code.....*  
} else {  
  // Sorry! No Web Worker support..  
}

create our web worker in an external JavaScript.

Here, we create a script that counts. The script is stored in the "workers.js" file:

postMessage() method - which is used to post a message back to the HTML page.

Now that we have the web worker file, we need to call it from an HTML page.

The following lines checks if the worker already exists, if not - it creates a new web worker object and runs the code in "demo\_workers.js":

if(typeof(w) == "undefined") {  
  w = new Worker("workers.js");  
}

Then we can send and receive messages from the web worker.

Add an "onmessage" event listener to the web worker.

w.onmessage = function(event){  
  document.getElementById("result").innerHTML = event.data;  
};

When the web worker posts a message, the code within the event listener is executed. The data from the web worker is stored in event. Data.

**SharedWorker**

Shared workers are special web workers that can be accessed by multiple browser contexts like browser tabs, windows, iframes, or other workers, etc.

The scripts that access the workers can do so by accessing it through the MessagePort object created using the SharedWorker.port property.

When the port is started multiple scripts can post messages to the worker and handle messages sent using the port.postMessage and port.onmessage respectively.

we have an onconnect handler assigned to the onconnect property. Then inside the handler function, we get the port that the shared worker uses to communicate with other scripts.

**ServiceWorkers**

Service workers essentially act as proxy servers that sit between web applications, the browser, and the network (when available). They are intended, among other things, to enable the creation of effective offline experiences, intercept network requests and take appropriate action based on whether the network is available, and update assets residing on the server. They will also allow access to push notifications and background sync APIs.

A service worker is an event-driven worker registered against an origin and a path. It takes the form of a JavaScript file that can control the web-page/site that it is associated with, intercepting and modifying navigation and resource requests, and caching resources in a very granular fashion to give you complete control over how your app behaves in certain situations (the most obvious one being when the network is not available).

In our Example serviceworker .html we are loading the main.js file and loading the serviceworker.js file using the command

navigator.serviceWorker.register("serviceworker.js")

After registered successfully it will display the status of the lifecycle whether installing or active or waiting by using query Selector from the html page

 window.addEventListener("message",function(event){}

we can sent the message to service worker and receives message

The Web Periodic Background Synchronization API provides a way to register tasks to be run in a [service worker](https://developer.mozilla.org/en-US/docs/Web/API/Service_Worker_API) at periodic intervals with network connectivity. These tasks are referred to as periodic background sync requests.

self.addEventListener('periodicsync', event => {

}

The Background Synchronization API provides a way to defer tasks to be run the service worker until the user has a stable network connection.

self.addEventListener('sync', event => {

if (event.tag == 'sync-messages') {

event.waitUntil(sendOutboxMessages());

}

});

The showNotification() method of the [ServiceWorkerRegistration](https://developer.mozilla.org/en-US/docs/Web/API/ServiceWorkerRegistration) interface creates a notification on an active service worker.

function showNotification() {

Notification.requestPermission(function(result) {

if (result === 'granted') {

navigator.serviceWorker.ready.then(function(registration) {

}

**Worklets:**

The Worklet interface is a lightweight version of Web Workers and gives developers access to low-level parts of the rendering pipeline.

With Worklets, you can run JavaScript an web Assembly code to do graphics rendering or audio processing where high performance is required.

The **AudioWorklet** interface of the Web Audio API is used to supply custom audio processing scripts that execute in a separate thread to provide very low latency audio processing.

The worklet's code is run in the Audio Worklet Global Scope global execution context, using a separate Web Audio thread which is shared by the worklet and other audio nodes.

In our Example we are accessing the microphone using getUserMedia.

We are creating new audioContext

Var audioCtx=new audioContext();

In audioCtx object we are creating audioworklet.addModule(TestModule.js) in promise we are sending our microphone input pass as a stream argument createMediaStreamSource and stored it in a variable source

And creating WorkletNode

 const worklet = n ew AudioWorkletNode(audioCtx, "TestWorklet");

first argument is audiocontex object and the second one is the processer name inside the Testwoeklet module. Afte that we are connecting audiocontex into destination(speaker) and start () or resume to display sound.

In ‘Test Worklet’ Module

We are creating a class that inherits the system-defined class AudioWorkletProcessor

class TestWorklet extends AudioWorkletProcessor {

    constructor(options) {

        super(options);

 process (inputs, outputs) {

 if (input[0])

        {

      this.port.postMessage(input[0]);

     }

Post Messages from microphone input

**Submitting Form**

**Html post Method:**

The form method=” post and action=”/userdet” method submits the form in the endpoint userdet in node js in index.js file

The node js uses the middleware app. app.use(bodyParser.json());to post the form data

**Ajax Method:**

XMLHttpRequest is a method used to submit the form data without refreshing the webpage

 xhttp.open("POST", "/userdet", true);

the object of XMLHttpRequest..open() method takes the first argument as get/post ,url,asynchronous true or false

and set the header

xhttp.setRequestHeader("Content-type", "application/x-www-form-urlencoded"); this.responseText;

                    console.log(this.responseText); will display the result from the server

**Jquery Post Method:**

Using Jquery when document is ready and on submitting form event

It will call the post method which takes URL, call back function inside data and status as arguments

 $. post("/userdet",{

                   uname:uname,

                   email:email

               },

               function (data, status)

it will display the data and the status as Success if successfully post the data

**Axios :**

We can use the URL

<script src="https://unpkg.com/axios/dist/axios.min.js"></script> inside our script to get the axios method

const response=await axios.post("/userdet",user)

post method takes first argument as url or endpoint second argument user as data and stored it in a variable response

The response.data and response.staus displays the data and staus respectively

**Fetch :**

fetch("/userdet",{

             method:'POST',

             body:JSON.stringify({

                 uname:uname,

                 email:email,

             }),

           headers:{

               'Content-type':'application/json; charset=UTF-8'

           }

         })

The fetch method takes the endpoint, method: post and body as data

It will return a promise. then(response) and. then(data) will display the response and data.

**sendBeacon :**

let result = navigator.sendBeacon(url,JSON.stringify(data));

The navigator .sendBeacon() method takes two arguments url and data.

l