Index

CSP:

Communications :

Client Side Storage:

- JavaScript variables
- DOM node storage
- Web Storage (localStorage and sessionStorage)
- IndexedDB
- Cache API
- File System Access API
- File and Directory Entries API
- cookies
- window.name
- WebSQL

Inputs:

- Types of Input
- Input values
- Hidden input
- Remove Inputs

Iframe:

- Creating iframe in javascript
- Hidden Iframe
- Minimized Iframe
- Open an Iframe
- SourceDocument
- Sandbox:
    - Loading Script
    - Formsubmit

    - Top-navigation
    - popups
- ChangeBackground
- ChangeContent

InsecuredDom:

- Client-xss

Javascript:

- Inline-script
- External-script
- Third-Party-script
- Nonce
- Sub Resource Integrity

Keyboard:

- KeyboardGlobal
- GetText-Value
- KeyboardEvent

Media:

- GeoLocations
- Webcamera/MicroPhone

StyleDocuments:

- Inline-style
- External-style

- Third-party-Style

Workers:

- WebWorkers
- Shared Workers
- Service-Workers
- Audio Worklets

Login form

UserName:Lava

Password:k9j8f48gkhlk4kjljj7dg66jhkhkkd

Routes:Localhost:3001/

Iframe

Change the Background:

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

Path of the file: C:\anu\Testbed\iframe\iframe.html

This is the index page of the iframe from here we can connect to various iframe options like loading iframe removing iframe

Create an Iframe Object

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

Path of the file: C:\anu\Testbed\iframe\loadingiframe.html

Removing an I frame:

We can access the I frame element by using **getElemetById ()** and remove it by using remove () method in JavaScript

Path of the file: C:\anu\Testbed\iframe\loadingiframe.html

Hidden Iframe:

- The hidden attribute hides the <iframe> element.
- We can specify either 'hidden' (without value) or 'hidden="hidden"'.
- A hidden <iframe> is not visible, but maintains its position on the page.

Path of the file: C:\anu\Testbed\iframe\hiddeniframe.html

Minimized I frame:

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

Path of the file: C:\anu\Testbed\iframe\minimizediframe.html

Transparent I frame:

We can make the I frame transparent by specifying the **opacity** CSS property to 0.

Path of the file: C:\anu\Testbed\iframe\transparentiframe.html

sandbox:

sandbox imposes set of restrictions to load the src file inside the I frame. If we specify the sandbox attribute empty it will apply all the policies and allow nothing to load

If you specify the flag name it will allow the specific directive

For example, if you want to allow the script file then you should

Use **sandbox=" allow-scripts"**

Allow the form submissions inside the frame by using the tag name and value

sandbox=" allow-forms"

sandbox=" allow -same-origin"

allows the link from the same origin

sandbox=" allow -popups"

allows the pop ups window to open

sandbox=allow-top-navigations

allows the window to be opened at the tap of the iframe

allows the link to load from the same origin.

Path of the file: C:\anu\Testbed\iframe\iframe.html

Content and Background:

We Can change the content and background colour of the frame

Input Methods

Create Input:

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

C:\anu\Testbed\input\inputform.html

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

C:\anu\Testbed\input\hiddeninput.html

Remove Input:

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

document. getElementById ("id "). remove ()

C:\anu\Testbed\input\deleteinputs.html

Get Text Value

Methods:

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

- getElementById ()
- getElementsByName ()
- getElementsByClassName ()
- getElementsByTagName ()
- document. query Selector()

C:\anu\Testbed\keyboard\gettextvalue.html

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

C:\anu\Testbed\javascript\index.html

C:\anu\Testbed\javascript\inlinescript.js

External Script:

We can create **script load ()** 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.

Nonce:

SubResource Inegrity

C:\anu\Testbed\javascript\main.js

Keyboard Event

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

C:\anu\Testbed\keyboard\keyboardevnt.html

Nonce:

A nonce is a random or semi-random number that is generated for a specific use. It is related to cryptographic communication and information technology (IT). The term stands for "number used once" or "number once" and is commonly referred to as a *cryptographic nonce*.

<script nonce=””>

</script>

SubResource Integrity:

Subresource Integrity (SRI) is a security feature that enables browsers to verify that resources they fetch (for example, from a CDN are delivered without unexpected manipulation. It works by allowing you to provide a cryptographic hash that a fetched resource must match.

Type this command in cmd prompt to create hashes

```
cat FILENAME.js | openssl dgst -sha384 -binary | openssl base64 -A
```

ou can use the following <script> element to tell a browser that before executing the https://example.com/example-framework.js script, the browser must first compare the script to the expected hash, and verify that there's a match.

<script src="https://example.com/example-framework.js"

integrity="sha384-oqVuAfXRKap7fdgcCY5uykM6+R9GqQ8K/uxy9rx7HNQlGYl1kPzQho1wx4JwY8wC"

crossorigin="anonymous"></script>

Key down Global:

After Pressing the key, the

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

C:\anu\Testbed\keyboard\keyboardglobal.html

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 country name state and city(data.counrty)

C:\anu\Testbed\media\geolocation.html

Web Camera & 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. **query Selector** we can access the video tag of HTML Element set the **srcattribute** as media stream and video. Play () will start the camera and microphone

C:\anu\Testbed\media\videoaudio.html

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(type of(w) == "undefined") {
  w = new Worker("workers.js");
}
```

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

Add an "on message" event listener to the web worker.

```
w. on message = function(event){
  document. getElementById("result"). inner HTML = 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.

C:\anu\Testbed\workers\webworkers\webworkers.html

SharedWorker

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

The scripts that access the workers can do so by accessing it through the **Message Port** 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. post Message and **port.onmess**age 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.

C:\anu\Testbed\workers\webworkers\sharedworkers.html

ServiceWorker:

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 **ServiceWorke**r .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 send 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 at periodic intervals with network connectivity. These tasks are referred to as periodic background sync requests.

self. AddEventListener ('periodic sync', 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. wait Until(sendOutboxMessages ());
  }
});
```

The show Notification () method of the ServiceWorkerRegistration interface creates a notification on an active service worker.

```
function show Notification () {
  Notification.requestPermission(function(result) {
    if (result === 'granted') {
      navigator. serviceWorker.ready. then(function(registration) {
}
```

C:\anu\Testbed\workers\serviceworker\serviceworker.html

C:\anu\Testbed\workers\serviceworker\main.js

C:\anu\Testbed\workers\serviceworker\serviceworker.js

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** a web Assembly code to do graphics rendering or audio processing where high performance is required.

The **Audio Worklet** 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 audio Context

Var audioCtx=new audioContext ();

In audioCtx object we are creating

audio worklet. 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 **Worklet Node**

const worklet = new AudioWorkletNode (audioCtx, "Test Worklet");

first argument is audiocontex object and the second one is the processer name inside the **Testwoeklet** module. After 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 Test Worklet extends AudioWorkletProcessor {

   constructor(options) {

      super(options);
```

```
process (inputs, outputs) {

if (input [0])

    {

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

      }
```

Post Messages from microphone input

C:\anu\Testbed\workers\worklets\audioworklet.html

Communications:

Html post Method:

The form method=" post and action="/userdet" method submits the form in the endpoint **userdet** in **node js**

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

C:\anu\Testbed\form\submitform.html

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. response Text;

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

C:\anu\Testbed\form\submitformajax.html

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 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

C:\anu\Testbed\form\jquerysubmitform.html

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’s display the data and status respectively

C:\anu\Testbed\form\formsubmitaxios.html

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.

Testbed\form\submitformfetchapi.html

send Beacon:

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

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

C:\anu\Testbed\form\submitformbeacon.html

WebSocket:

The WebSocket object provides the API for creating and managing a WebSocket connection to a server, as well as for sending and receiving data on the connection.

```
//connecting to  endpoint
    const ws=new WebSocket("ws:localhost:8081");
    //connection opened and submitting form data
    ws.addEventListener("open",function(eve){
    console.log("we are connected");
    ws.send(JSON.stringify({uname:uname, email:email}));

    });

    //postingback data from the server
    ws.addEventListener("message",function(eve){
       console.log(eve.data);
```

```
        document.getElementById("usn").innerHTML=`welcome
${eve.data}`;

    })
```

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

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

C:\Testbed\form\webscoket.html

Window-open:

The **open()** method of the Window interface loads a specified resource into a new or existing browsing context (that is, a tab, a window, or a iframe) under a specified name.

Window-open.html

```
form method="post" action="" name="parentform">

    <p>Enter username</p>

    <input type="text"  name="pname" placeholder="name">


  </form>

  <button onclick="openWindow()">Post</button>

  <script>

    function openWindow(){
```

```
        //alert(document.parentform.pname.value)

        window.open("/form/childwindow",width="300px", height="100px"
,left="100px");

        }
```

Childwindow.html

We are getting the value of input filed of a form from the parent document

```
 var username=opener.document.parentform.pname.value;
```

Insecured Dom:

What is DOM-based cross-site scripting?

DOM-based XSS vulnerabilities usually arise when JavaScript takes data from an attacker-controllable source, such as the URL, and passes it to a sink that supports dynamic code execution, such as eval() or innerHTML. This enables attackers to execute malicious JavaScript, which typically allows them to hijack other users' accounts.

To deliver a DOM-based XSS attack, you need to place data into a source so that it is propagated to a sink and causes execution of arbitrary JavaScript.

The most common source for DOM XSS is the URL, which is typically accessed with the window.location object. An attacker can construct **a** link to send a victim to a vulnerable page with a payload in the query string and fragment portions of the URL. In certain circumstances, such as when targeting a 404 page or a website running PHP, the payload can also be placed in the path.

For a detailed explanation of the taint flow between sources and sinks,

Client side Storage:

1. JavaScript variables
2. DOM node storage
3. Web Storage (localStorage and sessionStorage)
4. IndexedDB
5. Cache API
6. File System Access API
7. File and Directory Entries API
8. cookies
9. window.name
10. WebSQL

Storing and manipulating data in the browser — also known as client-side storage — is useful when it's not necessary or practical to send it to the web server.

Here are ten options for storing browser data:

Cache ApI:

We are Crearing a cache in the name of newCache In the async function we are passing the url of cache to open cahce and it will return the promise onsuccess display the messagecache created

```
const store=await cacheStore

("/clientsidestorage/data");

    console.log("cache stored");
```

```
    }
    async function cacheStore(url){
       try{
       const cache=await caches.open("newcache");
       await cache.add(url);
       //return true;
       Console.log("cache created");
}
```

same type of promise used to get the cache and we have created

```
 async function getCache(url){

    try{
      cache= await caches.open("newcache"),
       resp=await cache.match(url);
       return await resp.text();
    }
    catch(err)
    {
       return undefined;
       console.log(err);
```

```
    }
  }
```

Child Window:

Window.open method is used to open a new window

```
<form method="post" action="" name="parentform">
    <p>Enter username</p>
    <input type="text"  name="pname" placeholder="name">

  </form>

function openWindow(){
    window.open("/clientsidestorage/childwindow",width="300px", height="100px" ,left="100px");
    }
```

It will store the parent window variable and display it in child window using the following method in childwindow.html

Childwindow.html

```
 var username=opener.document.parentform.pname.value;
    document.write("Welcome "+ username);
```

}

Cookies:

Cookies are domain-specific data. They have a reputation for tracking people, but they're essential for any system which needs to maintain server state — such as logging on. Unlike other storage mechanisms, cookies are (usually) passed between the browser and server on *every* HTTP request and response. Both devices can examine, modify, and delete cookie data.

document.cookie sets cookie values in client-side JavaScript. You must define a string with a name and value separated by an equals symbol (=). In our Example**:**

we are getting the username using input and using two functions set cookie and getcookie to set and get the cookie values respectively.

```
document.cookie = "name="+uname;

{document.cookie}
```

File and Directory Entries API:

we are getting the username using input and

we are getting the username using input and using two functions set cookie and getcookie to set and

The File and Directory Entries API simulates a local file system that web apps can navigate around. You can develop apps that can read, write, and create files and directories in a sandboxed, virtual file system.

The File and Directory Entries API interacts with other related APIs. It was built on the File Writer API, which, in turn, was built on File API. Each of the APIs adds different functionality. These APIs are a giant evolutionary leap for web apps, which can now cache and process large amounts of data.

```
 window.requestFileSystem  = window.requestFileSystem ||
window.webkitRequestFileSystem;


   // Opening a file system with temporary storage

window.requestFileSystem(TEMPORARY, 1024*1024 /*1MB*/, (fs) => {


  fs.root.getFile('log.txt', {create: true, exclusive: false}, (fileEntry) => {

    console.log("creating file");

    fileEntry.isFile==true;

    fileEntry.name='log.txt';

    fileEntry.fullPath='/log.txt';

    console.log("File created");

    fileEntry.getMetadata((md)=>{

        console.log(md.modificationTime.toDateString());

    });
```

Note:Writngfile and blob methods are deprecated

MDN explicitly states: *do not use this on production sites*.

```
Create a FileWriter object for our FileEntry.

fileEntry.createWriter(function(fileWriter) {

console.log("writing file");

let text="new text";

var bb= new Blob([text]);

bb.append(text);
```

```
//let data = Blob([text], { type: "text/plain" });


//let data =  { type: "text/plain" });


fileWriter.write(bb.getBlob("text/plain"));
}, (fileError) => {
/* do whatever to handle the error */
console.log(error);
});


// fileWriter.onwrite = function(e) {
//     console.log(e);
//   console.log('Write completed.');
// };


// fileWriter.onerror = function(e) {
//   console.log('Write failed: ' + e.toString());
// };
// });


  });
});
}
```

File System Access API

The File System Access API allows a browser to read, write, modify, and delete files from your local file system. Browsers run in a sandboxed environment so the user must grant permission to a specific file or directory. This returns a

FileSystemHandle so a web application can read or write data like a desktop app.

let filehandle;

we can store the content entered inside the textarea in a file using the filehandle methods.

```
async function saveFile() {

    //get the value of textarea

    var cont = document.getElementById("cont").value;

    // create handle to a local file chosen by the user

    filehandle = await window.showSaveFilePicker();

    // create writable stream

    let stream = await filehandle.createWritable();

    // write the data

    await stream.write(cont);

     // save and close the file

    await stream.close();

    alert("file saved");
```

}

Indexed DB:

Indexed DB offers a NoSQL-like low-level API for storing large volumes of data. The store can be indexed, updated using transactions, and searched using asynchronous methods.

The IndexedDB API is complex and requires some event juggling. The following function opens a database connection when passed a name, version number, and optional upgrade function (called when the version number changes):

The following syntax open database in version 2

```
const request=indexedDB.open("empdb",2);
```

on success method will display the message the database created successfully

/success

```
request.onsuccess=function(e){
//alert("success called");
res=request.result;
console.log("success:"+res);
}
```

The objectStore is similar to table in sql

```
request.onupgradeneeded=function(event){
```

```
  var db=event.target.result;
  var objectstore=db.createObjectStore("employees",{keyPath:"id"});
  for(var i in emp)
    objectstore.add(emp[i]);
    console.log(objectstore);
 }
 //error
 request.error=function(e){
  console.log("Error");
 }
```

To add records we use the following method

```
 var req=res.transaction(["employees"],"readwrite")
  .objectStore("employees");
  req.add({id:newid,name:newname,job:newcity});
```

To read records

```
 var objectstore=res.transaction("employees").objectStore("employees");
  objectstore.openCursor().onsuccess=function(event){
      var cursor=event.target.result;
      if(cursor){
        //alert(cursor.key+cursor.value.name+cursor.value.job);
```

```
        //document.write(cursor.value.name);

        id.innerHTML+=cursor.key+"   "+cursor.value.name+ "
"+cursor.value.job +"<br>";



        console.log(cursor);

        cursor.continue();
```

JavaScript Variable:

Storing state in JavaScript variables is the quickest and easiest option.

Here we are using on object to store the details of an employee

```
var x=10;

   var emp={

   name:"Peter",

   age:30,

   job:"Developer",

   city:"Chennai",

   salary:50000

   }
```

```
document.getElementById("name").innerHTML="Name:"+emp.name;

document.getElementById("age").innerHTML= "Age:"+emp.age;

document.getElementById("job").innerHTML="Job:"+emp.job;

document.getElementById("city").innerHTML="City:"+emp.city;

document.getElementById("salary").innerHTML="Salary:"+emp.salary;
```

Dom Node Storage:

Most DOM elements, either on the page or in-memory, can store values in named attributes. It's safer to use attribute names prefixed with data-:

1. the attribute will never have associated HTML functionality
2. you can access values via a dataset property rather than the longer .setAttribute() and .getAttribute() methods.
3. Values are stored as strings so serialization and de-serialization may be

```
<div id="user" data-user-id="1" data-user-name="Simha" data-user-age="43" data-user-job>
```

We can access the data attributes using camelCase like this

```
id.innerHTML=user.dataset.userId;
```

Session Storage:

window.sessionStorage to retain session-only data while the browser tab remains open (but see Data Persistence)

Store or update named items with .setItem():

Like localstorage we can set and get session using

sessionStorage.setItem("name",name);

and

sessionStorage.getItem("name");

respectively

Enter User Name:

```
<script>
   function createSession(){
   var name= document.getElementById("uname").value;
   sessionStorage.setItem("name",name);
   var uname=sessionStorage.getItem("name");
   console.log(uname);
   document.getElementById("name").innerHTML="Welcome "+uname;
   }
</script>
```

Window.name:

Most The window.name property sets and gets the name of the window's browsing context. You can set a single string value which persists between browser refreshes or linking elsewhere and clicking back. For example:

```
function storeName(){
    var uname=document.getElementById("uname") .value;
   window.name=uname;
   console.log(window.name);
    }
```

Web SQL:

WebSQL was an effort to bring SQL-like database storage to the browser.

```
//(databaename.version.desc,size)
var db = openDatabase('mydb', '1.0', 'Test DB', 2 * 1024 * 1024);

db.transaction(function (tx) {
  tx.executeSql('CREATE TABLE IF NOT EXISTS LOGS (id unique, log)');
  tx.executeSql('INSERT INTO LOGS (id, log) VALUES (1, "foobar")');
  tx.executeSql('INSERT INTO LOGS (id, log) VALUES (2, "logmsg")');
});
```

```
db.transaction(function (tx) {
  tx.executeSql('SELECT * FROM LOGS', [], function (tx, results) {
    var len = results.rows.length, i;
    msg = "<p>Found rows: " + len + "</p>";
    document.querySelector('#status').innerHTML +=  msg;

    for (i = 0; i < len; i++) {
      alert(results.rows.item(i).log );
      document.querySelector('#status').innerHTML +=
results.rows.item(i).id+" ";
      document.querySelector('#status').innerHTML +=
results.rows.item(i).log+"<br>";

    }

  }, null);
});

    db.transaction(function(trans){
```

```
    console.log("Displaying Records");
      trans.executeSql("SELECT * FROM users",[],function(trans,res){
        console.log(res.rows);
        console.log("Displaying Records");
      });
    });
```

Content Security Policy:

The HTTP Content-Security-Policy response header allows web site administrators to control resources the user agent is allowed to load for a given page**.**

we have to create routes for corresponding directives in the routes folder

Scrip src:

We have to set header like this

res. set Header ("Content-Security-Policy", "script-src 'self')

it won’t allow the external or inline script

if you want to allow particular external script for example bootstrap inside our file then we have to specify the URL of the website.

Or else we can use the **unsafe-inline** or **nonce** or **hash** value inside our header

script-src: 'self' 'unsafe-inline'

Using a **nonce** is one of the easiest ways to allow the execution of inline scripts in a **Content Security Policy (CSP).** Here's how one might use it with the **CSP script-src directive**:

script-src 'nonce-rAnd0m';

we can specify some random value inside our script tag and we can use the same nonce value inside our res. setHeader method

res. setHeader ("Content-Security-Policy", "script-src 'self' 'nonce-bh3097129'");

<script nonce=" bh3097129">

</script>

It will check the nonce value will match the script nonce value if it so then it will allow the script to execute

http://localhost:3001/csp

There are corresponding links for script src directives and values

without CSP link will refer to

http://localhost:3001/csp/scriptsrc/?dir=scriptsrc&csp=off

it will take the req.query and stored it in a variable.

Var dir=req. query["dir"]

Var csp=req.qurery["csp"]

If CSP is off and the dir is script src there is no CSP set and it will allow the script to execute

http://localhost:3001/csp/scriptsrc/?dir=scriptsrc&csp=on

or else if the CSP is on then set the CSP header as script src self then it wont allow the inline or external script

likewise, we can fetch the dir from query **unsafe inline** and nonce and set the value

http://localhost:3001/csp/scriptsrc/?dir=unsafeinline&csp=on

http://localhost:3001/csp/scriptsrc/?dir=nonce&csp=on

Font src :

http://localhost:3001/csp/fontsrc

After loading this URL, it will display two links with CSP or without CSP

http://localhost:3001/csp/fontsrc/dir/?dir=fontsrc&csp=off

http://localhost:3001/csp/fontsrc/?dir=fontsrc&csp=on

if csp is off it will allow the font from google

if csp is on it won't allow the font from google

window. AddEventListener ('securitypolicyviolation', function (e) { // Add SecurityPolicyViolation event handler

console.log ({violated Directive: e. violated Directive, original Policy: e.originalPolicy });

The **securtiypolicyviolationevent** handler handles the **violationevent** and displays the policy directive and blocked URLTo display this we are storing this in partials directory **violationevent.ejs** it will display the details in the table

The same violationevnt .ejs can be used in scriptsrc , style src, imgsrc ,framesrc and fontsrc to display the violation event

```
var header = "font-src 'self'";
    var obj = {dir: dir, csp: csp, msg: "csp is off", header: header};
    res. setHeader ("Content-Security-Policy", "font-src 'self'");
    res. render ("csp/fontsrc", obj);
```

Frame src:

http://localhost:3001/csp/framesrc

```
 var header = "frame-src 'none'";
    var obj = {dir: dir, csp: csp, msg: msg, header: header };
    res.setHeader("Content-Security-Policy", "frame-src 'none'");
    res.render("csp/framesrc", obj);
```

Image src :

http://localhost:3001/csp/imgsrc

```
var obj = { dir: dir, csp: csp, msg: msg, header: "" };
  if (dir == "imgsrc" && csp == "off") {
    res. render("csp/imgsrc", obj);
```

Style src :

http://localhost:3001/csp/stylesrc

```
var header = "style-src 'self'";

var obj = {dir: dir, csp: csp, msg: msg, header: header};

  res. setHeader ("Content-Security-Policy", "style-src   'self' cdn.jsdelivr.net");

  res.render("csp/stylesrc", obj);
```

Worker src :

http://localhost:3001/csp/workersrc

```
var header = "worker-src 'none'";

var obj = {dir: dir, csp: csp, msg: msg, header: header };

  res.setHeader("Content-Security-Policy", "worker-src 'none' cdn.jsdelivr.net");

  res.render("csp/workersrc", obj);
```

Object src :

http://localhost:3001/csp/objectsrc

```
var header = "object-src 'none'";

  var obj = {dir: dir, csp: csp, msg: msg, header: header};

  res. setHeader ("Content-Security-Policy", "object-src 'none' cdn.jsdelivr.net");

  res.render("csp/objectsrc", obj);
```

Report-Only:

The HTTP **Content-Security-Policy-Report-Only** response header allows web developers to experiment with policies by monitoring (but not enforcing) their effects. These violation reports consist of JSON documents sent via an HTTP POST request to the specified URI.

res. setHeader ("Content-Security-Policy-Report-Only", "form-action 'none'; report-uri http://localhost:3001/csp/reporturi ");

it will send the report to the uri **csp/reporturi endpoint**. By using JavaScript **policyviolationevent** we can post the details using e. policy,e.documenturi etc to the report to end point using

form post method

<form action**="/csp/reporturi**" id="form2" method="post">

From the /reporturi endpoint it will render the ejs page sends the data by using the object cspdir and displays the data in reporturi.ejs

Navigate-To:

res.setHeader("Content-Security-Policy-Report-Only", "navigate-to 'none'”;

The HTTP Content-Security-Policy (CSP) **navigate-to** directive restricts the URLs to which a document can initiate navigations by any means including <form> (if form-action is not specified), **<a>, window.location, window.open,** etc. This is an enforcement on what navigations this document initiates, **not** on what this document is allowed to navigate to.

Using a <form> element with an action set to inline JavaScript will result in a CSP violation.

Sandbox:

The HTTP Content-Security-Policy (CSP) **sandbox** directive enables a sandbox for the requested resource similar to the <I frame> sandbox attribute. It applies restrictions to a page's actions including preventing popups, preventing the execution of plugins and scripts, and enforcing a same-origin policy.

In the csp/sandbox router it will render to the ejs page sandbox. ejs(The homepage of sandbox) It will display the table with three fields .When Without button clicked it will allow the resources There are two buttons withcsp if we specify the sandbox with empty value it won't allow the resources and send the violation error If we specify the sandbox with specific directives value it will allow the resources

allow-scripts:

If we set the header like this

res. setHeader("Content-Security-Policy", "sandbox");

it won't allow the script to execute

but if we specify the flag allow-scripts

res.setHeader("Content-Security-Policy", "sandbox allow-scripts");

it will allow the script to execute

allow-scripts:

Allows the content to be treated as being from its normal origin. If this keyword is not used, the embedded content is treated as being from a unique origin.

res.setHeader("Content-Security-Policy", "sandbox");

res.setHeader("Content-Security-Policy", "sandbox allow-same-origin");

allow-forms:

If we Allows the page to submit forms. If this keyword is not used, this operation is not allowed.

res.setHeader("Content-Security-Policy", "sandbox");

res.setHeader("Content-Security-Policy", "sandbox allow-forms");

allow-modals:

Allows the page to open modal windows.

res.setHeader("Content-Security-Policy", "sandbox");

res.setHeader("Content-Security-Policy", "sandbox allow-modals");

allow- top-navigation:

Allows the page to navigate (load) content to the top-level browsing context. If this keyword is not used, this operation is not allowed.

res.setHeader("Content-Security-Policy", "sandbox");

res.setHeader("Content-Security-Policy", "sandbox allow top-navigation");

allow- popups:

Allows popups (like from window.
open, target="_blank", showModalDialog). If this keyword is not used, that functionality will silently fail.

res.setHeader("Content-Security-Policy", "sandbox");

res.setHeader("Content-Security-Policy", "sandbox allow-popups");

allow- popups - to-escape-sandbox:

Allows a sandboxed document to open new windows without forcing the sandboxing flags upon them. This will allow, for example, a third-party advertisement to be safely sandboxed without forcing the same restrictions upon the page the ad links to.

res. setHeader ("Content-Security-Policy", "sandbox");

res. setHeader ("Content-Security-Policy", "sandbox allow-popups -to-escape-sandbox ");

allow- downloads:

Allows for downloads after the user clicks a button or link.

res.setHeader("Content-Security-Policy", "sandbox");

res.setHeader("Content-Security-Policy", "sandbox allow-downloads");

allow- downloads -without-user-activation:

Allows for downloads to occur without a gesture from the user.

res. setHeader("Content-Security-Policy", "sandbox");

res. setHeader("Content-Security-Policy", "sandbox allow-downloads without-user-activation");

pending works in testbed

serviceworker

input

navigate-to

audioworklets

sandbox =event violations