**1. Is Javascript single-threaded?**

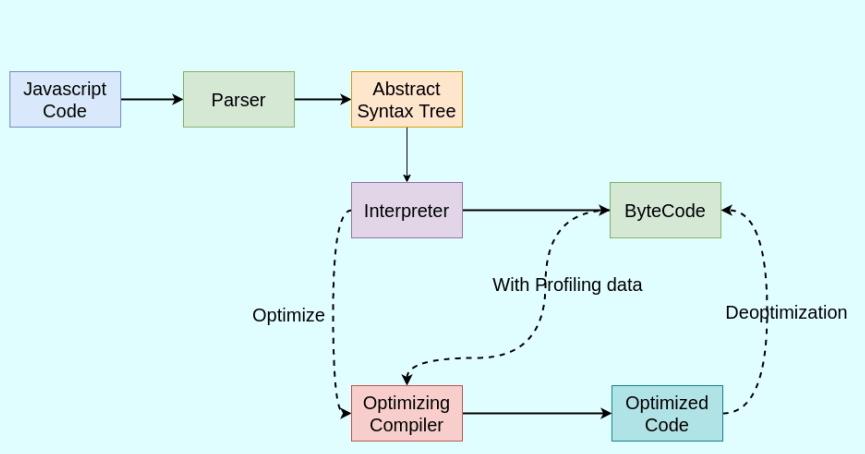
Yes, **JavaScript is** a **single-threaded language**. This means that it has **only one call stack and one memory heap**. Only one set of instructions is executed at a time.

Also, **Javascript is Synchronous and blocking** in nature. meaning that code is executed line by line and one task must be completed before the next one begins

However, JavaScript also has asynchronous capabilities, which allow certain operations to be executed independently of the main execution thread. This is commonly achieved through mechanisms like callbacks, promises, async/await, and event listeners. These asynchronous features enable JavaScript to handle tasks such as fetching data, handling user input, and performing I/O operations without blocking the main thread, making it suitable for building responsive and interactive web applications.

**2. Explain the main component of the JavaScript Engine and how it works.**

Every browser has a Javascript engine that executes the javascript code and converts it into machine code.



When JavaScript code is executed, the parser first reads the code and produces an AST, and stores it in memory. The interpreter then processes this AST and generates bytecode or machine code, which is executed by the computer.

The profiler is a component of a JavaScript engine that **monitors**the execution of the code.

Bytecode is used by optimizing the compiler along with profiling data. ”Optimizing compiler” or Just-in-time (JIT) compiler makes certain assumptions based on profiling data and produces highly optimized machine code.

Sometimes there is a case where the ‘optimization’ assumption is incorrect and then it goes back to the previous version via the “Deoptimize” phase (where it actually becomes the overhead to us)

JS Engine usually optimizes “hot functions” and uses inline caching techniques to optimize the code.

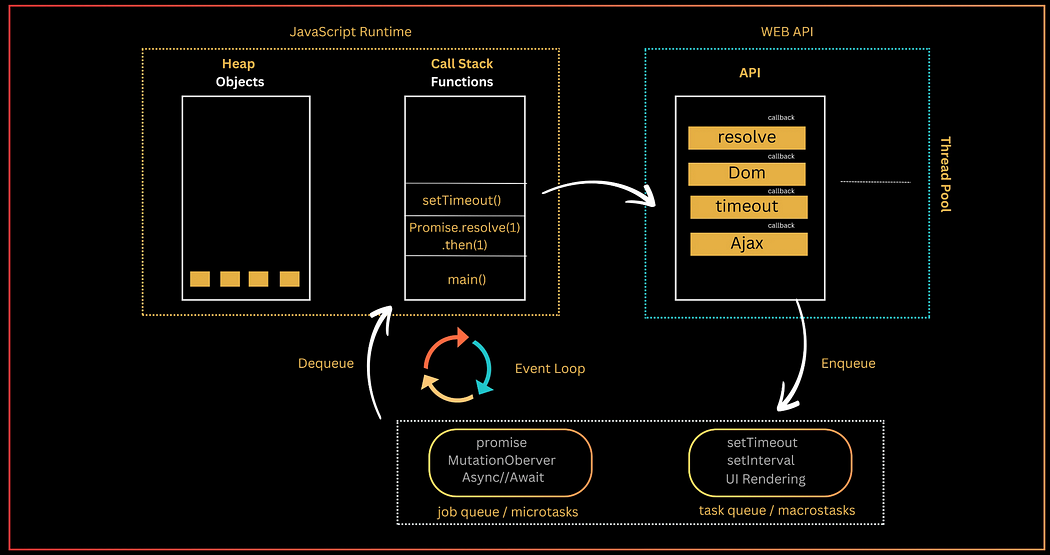
During this process, **the call stack keeps track of the currently executing functions, and the memory heap is used for memory allocation**.

Finally, the garbage collector comes into play to manage memory by reclaiming memory from unused objects.

**Google Chrome 𝗩𝟴 Engine:**

1. Interpreter is called **“Ignition”**.
2. Optimizing compiler is called **“TurboFan”**.
3. Apart from Parser, there is a “pre-parser” that checks for syntax and tokens
4. “**Sparkplug**” is introduced which is present between “Ignition” & “TurboFan” which is also called **Fast Compiler.**

**3. Explain the Event loop in JavaScript.**



The Event loop is a core component of the JavaScript runtime environment. It is **responsible for scheduling and executing asynchronous tasks**. The event loop works by continuously monitoring two queues: the call stack and the event queue.

**The call stack** is a **stack(LIFO) data structure** that stores the functions **that are currently being executed (**stores the execution context created during the code execution**)**.

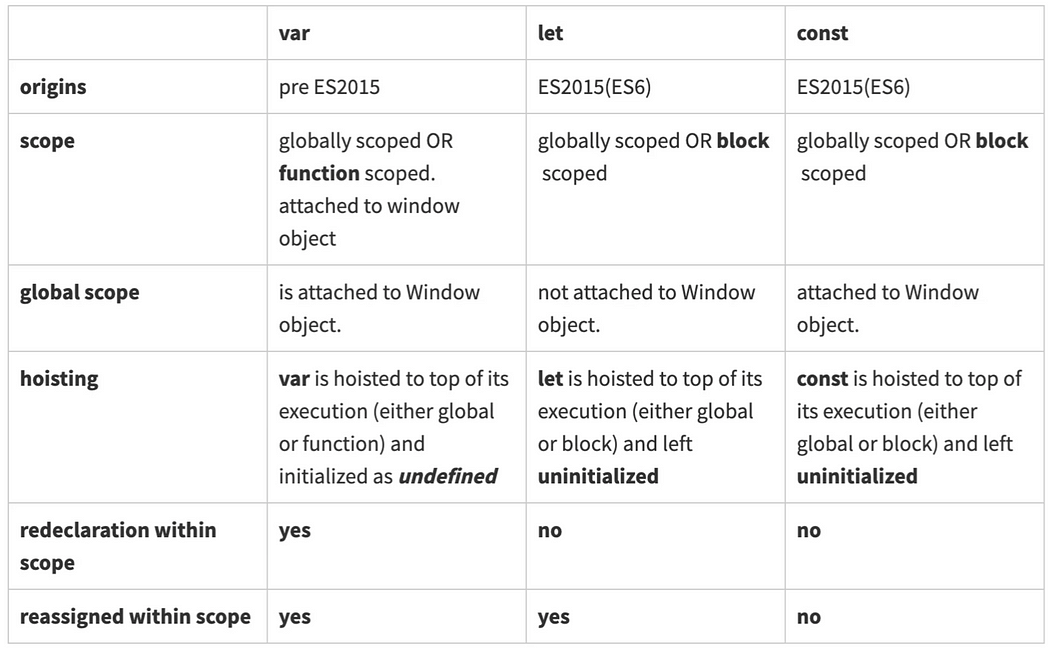
**Web APIs** is the place where the async operations (setTimeout, fetch requests, promises) with their callbacks are waiting to complete. **It borrows the thread from the thread pool to complete the task in the background** without blocking the main thread.

**The job queue (or microtasks)** is a FIFO (First In, First Out) structure that **holds the callbacks of async/await, promises, process.nextTick()** that are ready to be executed. For example, the resolve or reject callbacks of a fulfilled promise are enqueued in the job queue.

**The task queue (or macrostasks)** is a FIFO (First In, First Out) structure **that holds the callbacks of async operations (timer like setInterval, setTimeout)** that are ready to be executed. For example, the callback of a timed-out setTimeout() — ready to be executed — is enqueued in the task queue.

**The Event loop permanently monitors** whether the call stack is empty. If the call stack is empty, the event loop looks into the job queue or task queue and **dequeues any callback ready to be executed into the call stack.**

**4. Difference between var, let, and const?**



In a browser the **window object is the window of the browser**, the top structure in the HTML tree. **Variables declared with var globally are attached to the window object**. Type *var dog = ‘bowser’* in the browser’s console and then type *window.dog.*The value ‘bowser’ appears! This makes controlling the scope of the variable even more difficult. By contrast, **let and const**are not attached to the window object.

**6. What is callback function and callback hell ?**

In JavaScript, callbacks are commonly used to handle asynchronous operations.

**Callback function** is a function that is **passed as an argument** to another function and is intended **to be executed after the completion of a specific task or at a given time.**

function fetchData(url, callback) {  
 // Simulate fetching data from a server  
 setTimeout(() => {  
 const data = 'Some data from the server';  
 callback(data);  
 }, 1000);  
}  
  
function processData(data) {  
 console.log('Processing data:', data);  
}  
  
fetchData('https://example.com/data', processData);

In this example, the fetchData function takes a URL and a **callback function as arguments**. After fetching the data from the server (simulated using setTimeout), it calls the callback function and passes the retrieved data to it.

Callback Hell, also known as “Pyramid of Doom” is a term used in JavaScript programming to describe a situation where multiple nested callbacks are used within asynchronous functions.

“It occurs when asynchronous operations depend on the results of previous asynchronous operations, resulting in deeply nested and often hard-to-read code.”

Callback Hell is an anti-pattern with multiple nested Hcallbacks which makes code hard to read and debug when dealing with asynchronous logic.

function getData(callback) {

setTimeout(() => {

console.log('Data retrieved');

callback('data');

}, 1000);

}

function processData(data, callback) {

setTimeout(() => {

console.log('Processing data');

callback(data + ' processed');

}, 1000);

}

function saveData(processedData, callback) {

setTimeout(() => {

console.log('Data saved');

callback('done');

}, 1000);

}

// Example of callback hell

getData(function(data) {

processData(data, function(processedData) {

saveData(processedData, function(result) {

console.log(result);

});

});

});

In this example, each function (getData, processData, and saveData) performs an asynchronous operation and uses a callback to continue execution. The nested structure can quickly become difficult to manage and understand, especially as the number of asynchronous operations grows. This problem can be mitigated using modern JavaScript features like Promises or async/await.

**To avoid Callback Hell, modern JavaScript provides alternatives like Promises and async/await.**

**Using Promises**

Promises provide a way to handle asynchronous operations in a more readable and manageable way. Here's the previous example rewritten using Promises:

function getData() {

return new Promise((resolve, reject) => {

setTimeout(() => {

console.log('Data retrieved');

resolve('data');

}, 1000);

});

}

function processData(data) {

return new Promise((resolve, reject) => {

setTimeout(() => {

console.log('Processing data');

resolve(data + ' processed');

}, 1000);

});

}

function saveData(processedData) {

return new Promise((resolve, reject) => {

setTimeout(() => {

console.log('Data saved');

resolve('done');

}, 1000);

});

}

getData()

.then(processData)

.then(saveData)

.then(result => {

console.log(result);

})

.catch(error => {

console.error(error);

});

**Using async/await**

The async/await syntax, introduced in ES2017, allows you to write asynchronous code that looks more like synchronous code. This makes it easier to read and maintain. Here's the same example using async/await:

async function handleData() {

try {

const data = await getData();

const processedData = await processData(data);

const result = await saveData(processedData);

console.log(result);

} catch (error) {

console.error(error);

}

}

handleData();

In this version, async functions return a Promise, and await pauses the function execution until the Promise is resolved. This approach makes the code more linear and readable, avoiding deeply nested callbacks or .then() chains.

**7. What is Promise and Promise chaining?**

**Promise:**A Promise is **an object in JavaScript** used for asynchronous computations. **It represents the result of an asynchronous operation, the result may be resolved or rejected.**

Promises have three states:

1. **Pending**: The initial state. This is the state in which the Promise’s eventual value is not yet available.
2. **Fulfilled**: The state in which the Promise has been resolved successfully, and the eventual value is now available.
3. **Rejected**: The state in which the Promise has encountered an error or has been rejected, and the eventual value cannot be provided.

**Promise constructor** has two parameters**(resolve, reject)** **which are functions.** If the async task has been completed without errors then call the resolve function with message or fetched data to resolve the promise.

If an error occurred then call the reject function and pass the error to it.

we can access the result of promise using .then() handler.

we can catch the error in the .catch() handler.

// Creating a Promise  
const fetchData = new Promise((resolve, reject) => {  
 // Simulate fetching data from a server  
 setTimeout(() => {  
 const data = 'Some data from the server';  
 // Resolve the Promise with the retrieved data  
 resolve(data);  
 // Reject the Promise with an error  
 // reject(new Error('Failed to fetch data'));  
 }, 1000);  
});  
  
// Consuming the Promise  
fetchData  
 .then((data) => {  
 console.log('Data fetched:', data);  
 })  
 .catch((error) => {  
 console.error('Error fetching data:', error);  
 });

**Promise chaining:** The process of **executing a sequence of asynchronous tasks one after another using promises** is known as Promise chaining.

**It involves chaining multiple .then() methods**to a Promise to perform a series of tasks in a specific order.

new Promise(function (resolve, reject) {  
 setTimeout(() => resolve(1), 1000);  
})  
 .then(function (result) {  
 console.log(result); // 1  
 return result \* 2;  
 })  
 .then(function (result) {  
 console.log(result); // 2  
 return result \* 3;  
 })  
 .then(function (result) {  
 console.log(result); // 6  
 return result \* 4;  
 });

**8. What is async/await ?**

Async/await is a **modern approach** to handling asynchronous code in JavaScript. It provides a more concise and readable way to work with Promises and async operations, effectively avoiding the “Callback Hell” and improving the overall structure of asynchronous code.

In JavaScript, **the async keyword is used to define an asynchronous function, which returns a Promise**.

Within an async function, **the await keyword is used to pause the execution of the function until a Promise is resolved**, effectively allowing for synchronous-looking code while working with asynchronous operations.

async function fetchData() {  
 try {  
 const data = await fetch('https://example.com/data');  
 const jsonData = await data.json();  
 return jsonData;  
 } catch (error) {  
 throw error;  
 }  
}  
  
// Using the async function  
fetchData()  
 .then((jsonData) => {  
 // Handle the retrieved data  
 })  
 .catch((error) => {  
 // Handle errors  
 });

In this example, the fetchData function is defined as an async function, and it uses the await keyword to pause the execution and wait for the fetch and json operations, effectively working with Promises in a way that resembles synchronous code.

**12. What is a higher-order function?**

Higher-order function in JavaScript is **a function that** **either takes one or more functions as arguments or returns a function as its result.** In other words, it operates on functions, either by taking them as arguments, returning them, or both.

function operationOnArray(arr, operation) {  
 let result = [];  
 for (let element of arr) {  
 result.push(operation(element));  
 }  
 return result;  
}  
  
function double(x) {  
 return x \* 2;  
}  
  
let numbers = [1, 2, 3, 4];  
let doubledNumbers = operationOnArray(numbers, double);  
console.log(doubledNumbers); // Output: [2, 4, 6, 8]

They enable powerful techniques such as function composition, currying, and callback-based asynchronous operations. Understanding higher-order functions is essential for writing expressive and functional-style JavaScript code.

**An unary function (i.e. monadic) is a function that accepts exactly one argument.** It stands for a single argument accepted by a function.

**16. What is a prototype chain? and Object.create() method?**

**In JavaScript, every function and object has a property named prototype by default.**

Every object in JavaScript has a prototype. A prototype is another object from which the current object inherits properties and methods. You can think of the prototype as a template or a parent object.

**The prototype chain is a mechanism that allows objects to inherit properties and methods from other objects**

When you access a property or method on an object, JavaScript first looks for it on the object itself. If it doesn’t find it, it looks up the prototype chain until it finds the property or method. This process continues until it reaches the Object.prototype at the top of the chain.

**17. What is the difference between Call, Apply, and Bind methods?**

**Call:** The call() method invokes a function with a specified this value and **individual arguments passed as comma-separated values**

const person1 = { name: 'John' };  
 const person2 = { name: 'Jane' };  
  
 function greet(greeting) {  
 console.log(greeting + ' ' + this.name);  
 }  
  
 greet.call(person1, 'Hello'); // Output: Hello John  
 greet.call(person2, 'Hi'); // Output: Hi Jane

With call() method an object can use a method belonging to another object.

const o1 = {  
 name: 'ravi',  
 getName: function(){  
 console.log(`Hello, ${this.name}`)  
 }  
}  
  
const o2 = {  
 name: 'JavaScript Centric'  
}  
  
o1.getName.call(o2) // Hello, JavaScript Centric

**Apply:** Invokes the function with a given this value but **it accepts arguments as an array.** It is useful when the number of arguments to be passed is not known in advance or when the arguments are already in an array.

const numbers = [1, 2, 3, 4, 5];  
  
 const max = Math.max.apply(null, numbers);  
 console.log(max); // Output: 5

**bind:** instead of invoking it returns a new function and allows you to pass any number of arguments. bind() method takes an object as first argument and create a new function.

const module = {  
 x: 42,  
 getX: function() {  
 return this.x;  
 }  
 };  
  
  
 const boundGetX = unboundGetX.bind(module);  
 console.log(boundGetX()); // Output: 42

**20. What are the features of ES6?**

ES6, also known as ECMAScript 2015, introduced several new features and enhancements to JavaScript, significantly expanding the language’s capabilities. Some key features of ES6 include:

1. **Arrow Functions**
2. **Block-Scoped Variables**
3. **Classes**
4. **Modules**
5. **Template Literals**: Template literals allow for embedding expressions and multi-line strings using backticks, providing a more convenient way to create complex strings in JavaScript.
6. **Default Parameters**
7. **Rest and Spread Operators**
8. **Destructuring Assignment**
9. **Promises**
10. **Map, Set, WeakMap, WeakSet**: ES6 introduced new built-in data structures, such as Map and Set, for more efficient and specialized handling of collections and key-value pairs.
11. **Iterators and Generators**
12. **Enhanced Object Literals**

**21. What is Execution context, execution stack, variable object, and scope chain?**

**Execution Context:** the execution context refers to the environment in which a piece of code is executed. It consists of the scope, variable object, and the value of the “this” keyword.

Whenever a function is executed, an execution context is created and it contains all the variables or properties of that function.

There are three types of execution context in JavaScript:

> **Global Execution Context**

> **Functional Execution Context**

> **Eval Function Execution Context**

**Execution Stack:**It is also known as the “**call stack**,” a LIFO (Last in, First out) data structure that stores all the execution context of the function calls that are in progress. When a function is called, a new execution context is created and pushed onto the stack. When the function completes, its context is popped off the stack.

The engine executes the function whose execution context is at the top of the stack. When this function completes, its execution stack is popped off from the stack, and the control reaches the context below it in the current stack.

The execution context is created during the creation phase. The following things happen during the creation phase:

1. **LexicalEnvironment**component is created.
2. **VariableEnvironment**component is created.

**Variable Object:** It is a part of the execution context that contains all the variables, function declarations, and arguments defined in that context.

**Scope Chain:** The scope chain is a mechanism for resolving the value of a variable in JavaScript. When a variable is referenced, the JavaScript engine looks for the variable first in the current execution context’s variable object. If it’s not found there, it continues to the next outer execution context, following the scope chain, until it finds the variable or reaches the global execution context.

**22. What is the priority of execution of callback, promise, setTimeout, process.nextTick()?**

The priority of execution can be understood based on the event loop and the order in which different asynchronous operations are handled:

1. **process.nextTick():** Callbacks scheduled using process.nextTick() have the highest priority. When you use process.nextTick(), the callback is executed immediately after the current operation completes but before the event loop moves on to the next phase. This makes it a way to ensure that a function is executed at the earliest possible moment in the event loop.
2. **Promise:**Promises are typically executed after process.nextTick(). However, they are prioritized over callbacks scheduled with setTimeout().
3. **setTimeout():** Callbacks scheduled with setTimeout() are placed in the timer phase of the event loop. They will be executed after the current operation, promise, and any previously scheduled setTimeout() callbacks have been completed.
4. **Callback:** Regular callbacks (not scheduled using process.nextTick()) have the lowest priority. They are executed after the event loop processes process.nextTick(), promises, and setTimeout() callbacks.

**23. What is the Factory function and generator function?**

**A factory function** in JavaScript is a function **that returns an object**. It is a pattern used to create objects in a straightforward and organized manner. **Instead of using constructor functions and the new keyword** to create new objects, a **factory function encapsulates the object creation process and returns a new object.**

function createPerson(name, age) {  
 return {  
 name: name,  
 age: age,  
 greet: function() {  
 return `Hello, my name is ${this.name} and I am ${this.age} years old.`;  
 }  
 };  
}  
  
const person1 = createPerson('Alice', 25);  
const person2 = createPerson('Bob', 30);  
  
console.log(person1.greet()); // Output: Hello, my name is Alice and I am 25 years old.  
console.log(person2.greet()); // Output: Hello, my name is Bob and I am 30 years old.

**A Generator function** in JavaScript is a special type of function **that can be paused and resumed during its execution.**

A generator function produces a sequence of results instead of a single value.

When a generator function called **it returns a** **generator object**that can be used to control the execution of the function by calling the next() method.

The function’s code can be paused within the body using the **yield keyword**, and it can later be resumed from the exact point where it was paused.

function\* numberGenerator() {  
 let i = 0;  
 while (true) {  
 yield i++;  
 }  
}  
  
const gen = numberGenerator();  
console.log(gen.next().value); // Output: 0  
console.log(gen.next().value); // Output: 1  
console.log(gen.next().value); // Output: 2

This provides a powerful mechanism for creating iterators and handling asynchronous code.

**24. Different ways to clone (Shallow and deep copy of object) an object?**

**A shallow copy** is a copy of an object whose **references are the same** as the original object. This means that if you change the value of a property in the shallow copy, it will also change the value of the property in the original object.

const user = {  
 name: "Kingsley",  
 age: 28,  
 job: "Web Developer"  
}  
const clone = user

**A deep copy** is a copy of an object whose **references are not the same** as the original object. This means that if you change the value of a property in the deep copy, it will not change the value of the property in the original object.

there different ways to create deep copy of an object.

a)**JSON.parse and JSON.stringify**: useful for nested object also.

const originalObject = { name: "Alice", age: 25 };  
 const deepCopy = JSON.parse(JSON.stringify(originalObject));

b)**structuredClone:**

const myDeepCopy = structuredClone(myOriginal);

c)**Spread Operator(…):**any object with a nested object will not be deep copied.

const originalObject = { name: "Alice", age: 25 };  
const deepCopy = {...originalObject};  
  
deepCopy.name = "ravi"  
console.log("originalObject", originalObject.name) // Alice

d)**Object.assign():**the Object.assign() method should be used to deep copy objects that have no nested objects.

const originalObject = { name: "Alice", age: 25 };  
const shallowCopy = Object.assign({}, originalObject);

**25. How to make an object immutable? (seal and freeze methods)?**

In JavaScript, you can make an object immutable using the Object.seal() and Object.freeze() methods.

**Object.freeze()**: (**Completely Immutable)** this method freezes an object, making it both sealed and marking all its properties as read-only. After freezing an object, its properties cannot be modified, added, or removed.

const obj = { name: 'Alice', age: 25 };  
 Object.freeze(obj);  
 obj.name = 'Bob'; // Not allowed  
 obj.address = '123 Street'; // Not allowed  
 delete obj.age; // Not allowed

**Object.seal(): (Partialy Immutable)**this method seals an object, preventing new properties from being added and marking all existing properties as non-configurable. However, you can still modify the values of existing properties that are writable.

const obj = { name: 'Alice', age: 25 };  
 Object.seal(obj);  
 obj.name = 'Bob'; // Allowed  
 obj.address = '123 Street'; // Not allowed (no new properties can be added)  
 delete obj.age; // Not allowed (existing properties cannot be deleted)

**28. What are server-sent events?**

Server-sent events (SSE) are a simple and efficient technology for enabling **real-time updates from the server to the client**over a **single HTTP connection.**

SSE allows the server to push data to the web client (usually a browser) as soon as new information is available, making it an excellent choice for scenarios where you need real-time updates without relying on complex protocols or third-party libraries.

a)SSE provides a **unidirectional flow of data from the server to the client**. The server initiates the communication, sending updates to clients.

b)SSE **uses a text-based protocol,** which means that data sent from the server to the client is typically in a text format (usually JSON or plain text).

c)SSE handles reconnection automatically.

d)SSE establishes a persistent connection between the client and the server, allowing the server to send a stream of events to the client. Each event can have a unique type and data associated with it.

e)**The EventSource object is used to receive server-sent event notifications.** For example, you can receive messages from server as below,

if (typeof EventSource !== "undefined") {  
 var source = new EventSource("sse\_generator.js");  
 source.onmessage = function (event) {  
 document.getElementById("output").innerHTML += event.data + "<br>";  
 };  
}

f)Below are the list of events (onopen, onmessage,onerror) available for server-sent events.

**29. What is a web worker or service worker in javascript?**

Web Workers and Service Workers are two different concepts in JavaScript,

**Web Workers are designed for concurrent JavaScript execution** in the background, while **Service Workers are used for creating Progressive Web Apps with offline capabilities**and advanced features. Both are essential tools for enhancing the performance and functionality of web applications.

Each serves a distinct purpose in web development:

**Web Workers:**

1. **Concurrency:** **Web Workers are a browser feature** that allows you to run JavaScript code in the background, **separate from the main browser thread.** This enables concurrent execution of tasks without blocking the user interface.
2. **Use Cases:** Web Workers are commonly used for tasks that are computationally intensive or time-consuming, such as data processing, image manipulation, or complex calculations. By running these tasks in a separate thread, they don’t impact the responsiveness of the web page.
3. **Communication:** Web Workers can **communicate with the main thread using a messaging system**. They can send and receive messages, allowing for coordination between the main thread and the worker.
4. **Browser Support:** Web Workers are supported in most modern browsers.

**Service Workers:**

1. **Offline Capabilities:** Service Workers are a more advanced feature used for creating Progressive Web Apps (PWAs). They act as proxy servers that run in the background and can intercept and cache network requests. This enables offline capabilities, such as serving cached content when the user is offline.
2. **Use Cases:** Service Workers are primarily used for implementing features like **offline access,** **push notifications, and background sync.** They enable web apps to function even when there’s no internet connection.
3. Lifecycle: Service Workers have their own lifecycle with events like install, activate, and fetch. They are typically registered at the beginning of a web app's life.
4. Browser Support: Service Workers are supported in modern browsers and are a key technology for creating reliable and engaging web applications.

**30. How to compare 2 JSON objects in javascript?**

a) One simple way to compare two JSON objects is to use JSON.stringify to convert them into strings and then compare the strings.

function areEqual(obj1, obj2) {  
 return JSON.stringify(obj1) === JSON.stringify(obj2);  
}  
  
const obj1 = { a: 1, b: { c: 2 } };  
const obj2 = { a: 1, b: { c: 2 } };  
console.log(areEqual(obj1, obj2)); // Output: true

b) You can use the Ramda library to compare two JSON objects as well. Ramda provides a function called equals for this purpose.

const R = require('ramda');  
  
const obj1 = { a: 1, b: { c: 2 } };  
const obj2 = { a: 1, b: { c: 2 } };  
  
console.log(R.equals(obj1, obj2)); // Output: true

c) Another option is to use a library, such as Lodash, that provides a method for deep comparison of objects.

const \_ = require('lodash');  
  
const obj1 = { a: 1, b: { c: 2 } };  
const obj2 = { a: 1, b: { c: 2 } };  
console.log(\_.isEqual(obj1, obj2)); // Output: true