# General

## What is Nodejs

Node.js is a runtime environment that allows you to run JavaScript code on the server side, outside of a web browser. It is built on Chrome's V8 JavaScript engine, which makes it fast and efficient.

## What is a prototype chain

## What is the Temporal Dead Zone

The **Temporal Dead Zone (TDZ)** refers to the period in the code where a variable is declared using let or const but has not yet been initialized. During this period, any attempt to access the variable will result in a ReferenceError.

* When a block of code is entered, any variables declared with let or const within that block are "hoisted" to the top of their scope. However, unlike variables declared with var, they are not initialized.
* These variables remain uninitialized and in the TDZ until the line of code where they are initialized is reached.
* If you try to access the variable before it is initialized, a ReferenceError is thrown.

Why the TDZ Exists

The TDZ helps prevent certain types of bugs by ensuring that variables are not used before they are explicitly initialized, which can lead to unexpected behaviour. This feature enforces better coding practices and helps developers avoid subtle errors in their programs.

## What is Immediately Invoked Function Expression

An Immediately Invoked Function Expression (IIFE) is a JavaScript function that is executed as soon as it is defined. It’s a common design pattern used to create a new scope and avoid polluting the global scope, among other use cases.

(function() {

// Code inside this function

})();

OR

(function() {

// Code inside this function

}());

## What is memoization

**Memoization** is a programming optimization technique that involves storing the results of expensive function calls and reusing those stored results when the same inputs occur again. The primary goal of memoization is to speed up the execution of functions by avoiding redundant calculations.

**How Memoization Works**

1. **Cache Initialization**: A cache (usually an object or a dictionary) is used to store the results of function calls, where the keys are the function inputs, and the values are the computed results.
2. **Check Cache**: Before performing a computation, the function first checks if the result for the given input is already present in the cache.
3. **Return Cached Result**: If the result is in the cache, the function returns it immediately, avoiding the need to recompute it.
4. **Compute and Store**: If the result is not in the cache, the function computes it, stores the result in the cache, and then returns the result.

**Example in JavaScript**

Here’s a simple example of memoization in JavaScript using a recursive function to calculate the Fibonacci sequence:

function fibonacci(n, cache = {}) {

if (n <= 1) {

return n;

}

if (cache[n]) {

return cache[n];

}

cache[n] = fibonacci(n - 1, cache) + fibonacci(n - 2, cache);

return cache[n];

}

**Benefits of Memoization**

* **Improved Performance**: By avoiding repeated calculations, memoization can significantly reduce the time complexity of certain algorithms, especially those involving recursion or dynamic programming.
* **Efficient Resource Use**: Memoization optimizes resource use by saving computational time at the cost of increased memory usage for storing cached results.

**Use Cases**

* **Dynamic Programming**: Memoization is a key technique in dynamic programming to optimize problems that have overlapping subproblems, like the Fibonacci sequence, knapsack problem, etc.
* **Expensive Function Calls**: Memoization is useful in scenarios where function calls are expensive in terms of time or computational resources, and the same inputs are likely to occur multiple times.

**Limitations**

* **Memory Usage**: Since memoization involves storing results, it can lead to increased memory usage, which might be problematic for systems with limited memory.
* **Applicability**: Memoization is most effective for functions with deterministic results (i.e., the same inputs always produce the same outputs). Functions with side effects or non-deterministic behavior are not good candidates for memoization.

Top of Form

Bottom of Form

## What is Hoisting

**Hoisting** is a JavaScript behavior in which variable and function declarations are moved ("hoisted") to the top of their containing scope during the compile phase, before the code is executed. This means that variables and functions can be referenced before they are actually declared in the code, though the behavior can vary depending on how they are declared.

**How Hoisting Works**

* **Function Declarations**: Entire function declarations (both the name and the body) are hoisted to the top of their scope. This allows you to call a function before it is defined in your code.
* **Variable Declarations**: Variable declarations (with var, let, and const) are hoisted, but their initializations are not. For var, the variable is hoisted and initialized with undefined. For let and const, the variable is hoisted but remains uninitialized, resulting in a ReferenceError if you try to access it before its declaration.

## What are closures

## What are server-sent events

**Server-Sent Events (SSE)** is a web technology that allows a server to push real-time updates to the client over a single HTTP connection. Unlike WebSockets, which provide full-duplex communication, SSE is a one-way communication channel from the server to the client, making it ideal for scenarios where the server needs to send continuous updates to the client.

**How Server-Sent Events Work**

1. **Client Requests Updates**: The client (typically a web browser) initiates the connection by making an HTTP request to the server for updates.
2. **Server Sends Updates**: The server keeps the connection open and continuously sends data to the client whenever there are updates. The data is sent in plain text format, often in JSON, XML, or just text.
3. **Client Receives Updates**: The client listens for messages from the server and processes them as they arrive.

**Key Features of Server-Sent Events**

* **Text-Based Protocol**: SSE is a simple, text-based protocol, which makes it easy to implement and debug. The data is sent in a series of events, each consisting of lines of text.
* **Automatic Reconnection**: The client automatically reconnects to the server if the connection is lost, ensuring that updates are received continuously.
* **Event Handling**: SSE supports custom event types, allowing different types of data to be handled in different ways on the client side.
* **Built-In Support in Browsers**: Most modern browsers support SSE natively through the EventSource API.

**Example of Server-Sent Events**

**Server-Side (Node.js Example)**

javascript

Copy code

const http = require('http');

http.createServer((req, res) => {

if (req.url === '/events') {

res.writeHead(200, {

'Content-Type': 'text/event-stream',

'Cache-Control': 'no-cache',

'Connection': 'keep-alive'

});

setInterval(() => {

res.write(`data: ${new Date().toISOString()}\n\n`);

}, 1000);

} else {

res.writeHead(404);

res.end();

}

}).listen(3000);

**Explanation**:

* The server listens for requests on port 3000. If the client requests /events, the server responds with an SSE stream, sending the current date and time every second.

**Client-Side (HTML/JavaScript Example)**

html

Copy code

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

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

<title>Server-Sent Events</title>

</head>

<body>

<div id="updates"></div>

<script>

const eventSource = new EventSource('/events');

eventSource.onmessage = function(event) {

document.getElementById('updates').innerHTML += `Update: ${event.data}<br>`;

};

eventSource.onerror = function(event) {

console.error('Error with event source', event);

};

</script>

</body>

</html>

**Explanation**:

* The EventSource object in JavaScript is used to create a connection to the server's /events endpoint.
* The onmessage event handler is triggered whenever the server sends a message, and it updates the page with the data.

**Use Cases for Server-Sent Events**

1. **Live Notifications**: SSE is ideal for pushing live notifications or updates, such as social media feeds, stock prices, or news updates.
2. **Real-Time Monitoring**: SSE can be used for real-time monitoring dashboards, where the server needs to continuously send status updates to the client.
3. **Chat Applications**: While WebSockets are typically used for chat applications, SSE can be used when only one-way communication is needed, such as broadcasting messages to all connected clients.

**Advantages of SSE**

* **Simple to Implement**: SSE is simpler to implement than WebSockets for scenarios where only one-way communication is needed.
* **Automatic Reconnection**: The client automatically reconnects if the connection is dropped, making it reliable for long-running connections.
* **Works Over HTTP/HTTPS**: SSE works over standard HTTP/HTTPS, so it is compatible with existing web infrastructure, including proxies and firewalls.

**Limitations of SSE**

* **One-Way Communication**: SSE only supports one-way communication from the server to the client. For bi-directional communication, WebSockets are more appropriate.
* **Limited Browser Support**: While most modern browsers support SSE, older browsers might not. However, it is generally well-supported.
* **Scalability**: Since SSE keeps connections open, it can be less scalable than stateless HTTP requests, especially if there are many connected clients.

## Why do you need strict mode

**Key Reasons to Use Strict Mode**

1. **Catches Common Coding Mistakes**: Strict mode helps catch errors that might otherwise go unnoticed, improving code quality and reducing bugs.
2. **Prevents the Use of Undeclared Variables**: In non-strict mode, assigning a value to an undeclared variable automatically creates it as a global variable. Strict mode throws an error when trying to assign a value to a variable that has not been declared, preventing accidental global variable creation.

"use strict";

myVariable = 10; // ReferenceError: myVariable is not defined

1. **Eliminates this Binding for Global Objects**: In non-strict mode, if you call a function without an object context, this refers to the global object (window in browsers). In strict mode, this remains undefined, preventing unintended global variable access.

"use strict";

function test() {

console.log(this); // undefined

}

test();

1. **Disallows Duplicate Parameter Names**: Strict mode prevents the use of duplicate parameter names in function declarations, which can lead to unexpected behavior.

"use strict";

function test(a, a) { // SyntaxError: Duplicate parameter name not allowed in this context

return a;

}

1. **Throws Errors on Assignment to Non-Writable Properties**: Strict mode throws an error when trying to assign a value to a non-writable property, a getter-only property, or a non-existent property on an object that is not extensible.

"use strict";

const obj = {};

Object.defineProperty(obj, 'x', { value: 42, writable: false });

obj.x = 9; // TypeError: Cannot assign to read only property 'x' of object

1. **Prevents Deletion of Non-Configurable Properties**: In strict mode, deleting non-configurable properties (like built-in objects' properties) results in an error.

"use strict";

delete Object.prototype; // TypeError: Cannot delete property 'prototype' of function Object() { [native code] }

1. **Secures eval and arguments**: Strict mode introduces changes to eval() and arguments, making them safer:
   * Variables declared inside eval do not leak into the surrounding scope.
   * arguments no longer reflects changes made to function parameters.
2. **Reserves Future Keywords**: Strict mode reserves certain keywords (like implements, interface, let, package, private, protected, public, static, yield) for future use, helping to ensure forward compatibility.

"use strict";

let implements = 1; // SyntaxError: Unexpected strict mode reserved word

**How to Enable Strict Mode**

* **Globally** (for the entire script):

"use strict";

// Your code here

* **Locally** (within a specific function):

function myFunction() {

"use strict";

// Strict mode is applied here

}

**Use Cases for Strict Mode**

* **Legacy Code**: When maintaining or refactoring older JavaScript codebases, strict mode helps identify and fix potential issues.
* **Best Practices**: Adopting strict mode from the start encourages best practices and helps avoid common pitfalls.
* **Security**: By enforcing stricter rules, strict mode can help protect against certain security vulnerabilities.

**Summary**

Strict mode is an important tool for writing cleaner, safer, and more efficient JavaScript code. By opting into strict mode, developers can catch errors early, prevent unintended behaviors, and ensure that their code adheres to modern JavaScript standards.

Bottom of Form

## What is event bubbling

## How do you generate random integers

function getRandomInt(min, max) {

min = Math.ceil(min);

max = Math.floor(max);

return Math.floor(Math.random() \* (max - min + 1)) + min;

}

## What is the purpose of freeze method

The Object.freeze() method in JavaScript is used to freeze an object, which means that the object can no longer be modified in any way. This includes adding, removing, or changing properties and their values. Once an object is frozen, it becomes immutable.

**Key Features of Object.freeze()**

1. **Prevent Property Modification**: After an object is frozen, you cannot change the values of its existing properties.

const obj = Object.freeze({ name: "Alice" });

obj.name = "Bob"; // This will not work, name remains "Alice"

console.log(obj.name); // Output: "Alice"

1. **Prevent New Properties from Being Added**: You cannot add new properties to a frozen object.

const obj = Object.freeze({ name: "Alice" });

obj.age = 30; // This will not work, no new properties can be added

console.log(obj.age); // Output: undefined

1. **Prevent Properties from Being Removed**: You cannot delete properties from a frozen object.

const obj = Object.freeze({ name: "Alice" });

delete obj.name; // This will not work, name cannot be deleted

console.log(obj.name); // Output: "Alice"

1. **Prevent Reconfiguration of Properties**: You cannot reconfigure the property descriptors (like making a property non-enumerable or non-writable) on a frozen object.

const obj = Object.freeze({ name: "Alice" });

Object.defineProperty(obj, 'name', { writable: true }); // Throws a TypeError in strict mode

**Example of Object.freeze()**

javascript

Copy code

const person = {

name: "John",

age: 30

};

Object.freeze(person);

person.name = "Jane"; // This will not work, name remains "John"

delete person.age; // This will not work, age remains 30

person.gender = "male"; // This will not work, no new properties can be added

console.log(person); // Output: { name: "John", age: 30 }

**Use Cases for Object.freeze()**

1. **Immutable Data Structures**: Object.freeze() is commonly used in scenarios where you want to create immutable data structures, ensuring that certain objects remain constant throughout the program and cannot be altered.
2. **Constants**: It can be used to create constant objects whose properties should not be changed after they are set.
3. **Security**: In environments where security is a concern, Object.freeze() can be used to protect critical parts of an object from accidental or malicious modification.
4. **Preventing Changes in Libraries or Frameworks**: When using third-party libraries or frameworks, freezing objects can prevent them from modifying critical parts of your application.

**Important Considerations**

* **Shallow Freeze**: Object.freeze() only performs a shallow freeze. This means that if the object contains other objects (nested objects), the nested objects are not frozen and can still be modified.

const obj = Object.freeze({

name: "Alice",

details: { age: 25 }

});

obj.details.age = 30; // This works, as `details` is not frozen

console.log(obj.details.age); // Output: 30

To deeply freeze an object (including all nested objects), you would need to recursively freeze each object:

function deepFreeze(obj) {

Object.freeze(obj);

Object.keys(obj).forEach(key => {

if (typeof obj[key] === 'object' && obj[key] !== null) {

deepFreeze(obj[key]);

}

});

return obj;

}

const person = deepFreeze({

name: "John",

details: { age: 30 }

});

person.details.age = 35; // This will not work, as `details` is now frozen

console.log(person.details.age); // Output: 30

## What is V8 JavaScript engine

The **V8 JavaScript engine** is an open-source JavaScript engine developed by Google. It is designed to execute JavaScript code efficiently and is primarily used in the Google Chrome browser and in Node.js, among other environments. V8 is known for its high performance, modern features, and extensive optimizations.

**Key Features of V8**

1. **Just-In-Time (JIT) Compilation**: V8 uses JIT compilation to improve the execution speed of JavaScript code. It compiles JavaScript code into native machine code at runtime, rather than interpreting it line-by-line. This allows for faster execution compared to traditional interpreters.
2. **Optimizing Compiler**: V8 includes an optimizing compiler called *Ignition* and a subsequent *Turbofan* compiler. Ignition compiles JavaScript to bytecode, while Turbofan performs further optimizations on the bytecode to generate highly optimized machine code.
3. **Garbage Collection**: V8 uses a garbage collector to automatically manage memory. It identifies and cleans up unused objects to prevent memory leaks and optimize memory usage.
4. **Inline Caching**: V8 uses inline caching to speed up property access and function calls by caching information about previously accessed properties and functions.
5. **Hidden Classes**: V8 employs hidden classes to optimize object property access. This technique allows V8 to maintain efficient access paths to object properties by creating hidden classes that reflect the structure of the object.
6. **Concurrency Support**: While JavaScript is single-threaded, V8's architecture and optimizations enable efficient handling of asynchronous tasks and events, which is crucial for applications like web servers and real-time applications.
7. **Optimized for Modern JavaScript**: V8 supports the latest ECMAScript standards and features, providing support for modern JavaScript syntax and functionalities.

**Example Usage**

* **Google Chrome**: V8 is the core JavaScript engine used in Google Chrome, where it interprets and executes JavaScript code in web pages.
* **Node.js**: V8 is also used in Node.js, a server-side JavaScript runtime. In this environment, V8 executes JavaScript code to build server applications.

**How V8 Works**

1. **Parsing**: When JavaScript code is loaded, V8 parses the code into an abstract syntax tree (AST).
2. **Bytecode Compilation**: The AST is converted into bytecode by the Ignition interpreter.
3. **Execution and Optimization**: The bytecode is executed, and V8 collects runtime information. This information is used to optimize the code further. The optimized code is compiled into machine code by the Turbofan compiler.
4. **Garbage Collection**: V8 periodically runs a garbage collector to reclaim memory from objects that are no longer in use.

**Advantages of V8**

* **Performance**: V8's combination of JIT compilation and optimization techniques makes it one of the fastest JavaScript engines available.
* **Modern Features**: V8 supports the latest JavaScript features, providing developers with advanced capabilities and improved performance.
* **Cross-Platform**: V8 is designed to work across different platforms and environments, making it versatile and widely used.

## What is destructuring assignment

**Practical Use Cases**

1. **Function Parameters**: Destructuring is often used in function parameters to simplify code and make it more readable.

function printPerson({ name, age }) {

console.log(`Name: ${name}, Age: ${age}`);

}

printPerson({ name: 'Alice', age: 30 });

1. **State Management**: In libraries like React, destructuring is frequently used to access and manage state and props.

const [count, setCount] = useState(0);

1. **Returning Multiple Values**: Destructuring is useful when a function returns multiple values or an object with multiple properties.

function getCoordinates() {

return [10, 20];

}

const [x, y] = getCoordinates();

## What are streams

## What is JWT

## What is the difference between for, foreach, map, filter and reduce

**1. for Loop**

**Description**: The traditional for loop provides maximum control over the iteration process. It's suitable for performance-critical scenarios where fine-tuned control is necessary.

**Syntax**:

for (let i = 0; i < array.length; i++) {

// code to execute

}

**Performance Considerations**:

* **Performance**: Typically the fastest among these methods due to its simplicity and lack of overhead. There are no function calls or additional abstractions.
* **Flexibility**: Allows fine control over iteration, including early exit (break), continuing to the next iteration (continue), and manual indexing.
* **Readability**: More verbose compared to modern iteration methods but provides maximum control.

**Example**:

const numbers = [1, 2, 3, 4, 5];

for (let i = 0; i < numbers.length; i++) {

console.log(numbers[i]);

}

**2. forEach()**

**Description**: The forEach() method is an array method that executes a provided function once for each array element.

**Syntax**:

array.forEach((element, index, array) => {

// code to execute

});

**Performance Considerations**:

* **Performance**: Generally slower than traditional for loops due to the overhead of function calls for each iteration.
* **Flexibility**: Cannot be broken out of or returned from early; always iterates through all elements.
* **Readability**: Offers a clean and functional approach but with less control over iteration flow.

**Example**:

const numbers = [1, 2, 3, 4, 5];

numbers.forEach(number => {

console.log(number);

});

**3. for...of**

**Description**: The for...of loop is used to iterate over iterable objects such as arrays, strings, maps, and sets.

**Syntax**:

for (const element of iterable) {

// code to execute

}

**Performance Considerations**:

* **Performance**: Usually faster than forEach() and competitive with for loops, as it doesn't involve function calls.
* **Flexibility**: Supports early exit with break, continuing with continue, and returning early with return.
* **Readability**: Clean and straightforward for iterating over iterable objects.

**Example**:

const numbers = [1, 2, 3, 4, 5];

for (const number of numbers) {

console.log(number);

}

**4. for...in**

**Description**: The for...in loop is used to iterate over enumerable properties of an object.

**Syntax**:

for (const key in object) {

// code to execute

}

**Performance Considerations**:

* **Performance**: Slower compared to other methods, particularly for arrays, due to property enumeration and potential inclusion of inherited properties.
* **Flexibility**: Useful for iterating over object properties but less suitable for arrays due to possible issues with property enumeration.
* **Readability**: Best used for objects, but care must be taken to avoid iterating over prototype properties.

**Example**:

const person = { name: 'Alice', age: 30 };

for (const key in person) {

if (person.hasOwnProperty(key)) { // Check to avoid inherited properties

console.log(key, person[key]);

}

}

**Summary**

* **for Loop**: Fastest and most flexible. Best for performance-critical tasks where fine-grained control is needed.
* **forEach()**: Slower due to function call overhead but provides a clean, functional approach. Not suitable if you need to break or return early.
* **for...of**: Efficient and straightforward for iterating over iterables. Supports early exits and is generally faster than forEach().
* **for...in**: Useful for iterating over object properties. Slower and less suitable for arrays; includes enumerable properties and can iterate over prototype chain properties.

Choosing the right iteration method depends on the specific needs of your code, such as performance requirements, control over iteration, and readability. For most array operations, for...of and traditional for loops offer a good balance of performance and readability. For objects, for...in is suitable but should be used with care.

Bottom of Form

## Event Loop Can you explain in detail the phases of the Node.js event loop? Describe what happens in each phase.

## What is the event loop in Node.js, and why is it important?

## What happens in the timers phase of the event loop?

## What is the difference between setImmediate() and setTimeout in Node.js?

## How can you prevent the event loop from being blocked?

## What are micro and macro task queues?

# Memory Management

## What is a memory leak and how to detect and diagnose it.

## What is the process.memoryUsage() method, and what information does it provide?

## What is the difference between stack memory and heap memory in Node.js?

## How does the V8 engine handle garbage collection.

## What is buffer in Node.js

## Why is it important to avoid using global variables

# Clustering

## What is Node.js clustering

## Can you describe a scenario where Node.js clustering might not be the best solution for scaling an application?

## What are the advantages and disadvantages of clustering

## For what purpose we will use OS module for clustering

## What is the default load balancer is being used for clusting