javascript jd based

1. JavaScript Basics

✅ 1. What is JavaScript, and how is it different from Java?

■ Explanation (simple, conceptual)

JavaScript is a lightweight, interpreted programming language mainly used for building interactive web pages. It runs in browsers and also on servers (Node.js).

Java is a compiled, general-purpose, object-oriented language used for backend systems, enterprise apps, Android, etc.

■ When/Why is it used

- JavaScript → frontend interactivity, DOM manipulation, APIs, event handling
- Java → backend microservices, enterprise apps, financial systems, large-scale platforms

■ Example

JavaScript code:

```
console.log("Hello JS");
```

Java code:

```
System.out.println("Hello Java");
```

■ Short interview line

"JavaScript is an interpreted, dynamic, browser-based language, while Java is a compiled, strongly typed, backend-focused language."

✅ 2. Explain the difference between var, let, and const

■ Explanation (simple, conceptual)

Keyword	Scope	Reassign?	Redeclare?	Hoisted?
var	Function scope	Yes	Yes	Yes (initialized as undefined)
let	Block scope	Yes	No	Hoisted but not initialized (TDZ)
const	Block scope	No	No	Hoisted but not initialized (TDZ)

■ When/Why is it used

- **var** → old JS, avoid using
- **let** → variables that change
- **const** → constants, or objects/arrays whose reference does not change

■ Example

```
var x = 10;
let y = 20;
const z = 30;
```

■ Short interview line

"var is function-scoped, let and const are block-scoped; const cannot be reassigned."

✅ 3. What is hoisting in JavaScript?

■ Explanation (simple, conceptual)

Hoisting means JavaScript moves variable and function declarations to the **top of their scope** during the compile phase.

- For `var`: hoisted but set to `undefined`
- For `let` and `const`: hoisted but **not initialized** → in the Temporal Dead Zone (TDZ)
- For functions: entire function is hoisted

■ When/Why is it used

Understanding hoisting avoids unexpected behavior when variables appear to be used before declaration.

■ Example

```
console.log(a);  // undefined
var a = 10;

console.log(b);  // ReferenceError (TDZ)
let b = 20;
```

■ Short interview line

“Hoisting lifts declarations to the top, but only var is initialized; let/const stay in the temporal dead zone.”

✅ 4. What are data types in JS? (Primitive vs Object)

■ Explanation (simple, conceptual)

JavaScript has **two categories** of data types:

1. Primitive Types (immutable)

- Number
- String
- Boolean
- Null
- Undefined
- Symbol
- BigInt

2. Object Types (reference types)

- Objects
- Arrays
- Functions
- Dates, Maps, Sets, etc.

■ When/Why is it used

- Primitives → stored by value
- Objects → stored by reference, allows collections & complex structures

■ Example

```
let x = 10;          // primitive
let obj = {a: 1};    // object
```

■ Short interview line

“JS has primitives (stored by value) and objects (stored by reference).”

✅ 5. What is the difference between `==` and `===`?

■ Explanation (simple, conceptual)

- **== (loose equality)**

  Performs **type conversion** before comparing

- **=== (strict equality)**

  Compares **both value and type**, no conversion

■ When/Why used

Use **===** always in modern JavaScript to avoid unexpected conversions.

■ Example

```
5 == "5"   // true (type conversion)
5 === "5"  // false (different types)
```

■ Short interview line

“== does type conversion while === checks value and type strictly.”

---

2. Functions

✅ 1. What are function declarations vs function expressions?

■ Explanation (simple, conceptual)

Function Declaration

- A standalone function defined with the `function` keyword.
- Fully **hoisted** → can be used before definition.

Function Expression

- A function assigned to a variable.
- **Not hoisted** like declarations (exists only after runtime assignment).

■ When/Why is it used

- Use **declarations** for reusable functions you want available globally.
- Use **expressions** for inline logic, callbacks, or when you want function-as-a-value.

■ Example

```
// Function Declaration
function add(a, b) {
  return a + b;
}

// Function Expression
const multiply = function(a, b) {
  return a * b;
};
```

■ Short interview line

"Declarations are hoisted and defined with `function`; expressions assign a function to a variable and are not hoisted the same way."

✅ 2. What is an arrow function? How is it different from a normal function?

■ Explanation (simple, conceptual)

Arrow functions are a shorter syntax for writing functions introduced in ES6.

■ Key differences

- **No own** `this` → inherits `this` from surrounding scope (lexical this)
- **Cannot be used as constructors** (`new` doesn't work)
- **No** `arguments` **object**
- **Shorter, cleaner syntax**

■ When/Why is it used

- Useful in callbacks, array operations (map, filter)
- Best when you want to avoid binding `this` manually
- Great for concise one-line functions

■ Example

```
// Normal function
function greet() {
  console.log(this);
}

// Arrow function
const greetArrow = () => {
  console.log(this);
};
```

■ Short interview line

“Arrow functions don’t have their own `this` or `arguments`, can't be constructors, and provide a shorter syntax.”

✅ 3. What is a callback function?

■ Explanation (simple, conceptual)

A **callback function** is a function passed as an argument to another function and executed later.

■ When/Why is it used

- For asynchronous tasks (API calls, timers)
- For array operations (map, filter, reduce)
- For event handling

■ Example

```
function greet(name, callback) {
  callback(name);
}

greet("Mohini", function(n) {
```

```
  console.log("Hello " + n);
});
```

■ Short interview line

“A callback is a function passed to another function to be executed later, especially in async programming.”

✅ 4. What is a higher-order function?

■ Explanation (simple, conceptual)

A **higher-order function (HOF)** is a function that:

- Takes another function as input

  OR

- Returns another function

■ When/Why is it used

- Functional programming
- Reusable logic
- Array operations (`map`, `filter`, `reduce`)
- Middleware patterns

■ Example

```
function higherOrder(func) {
  return function(x) {
    return func(x);
  };
}

const double = n => n * 2;
const result = higherOrder(double);
console.log(result(10)); // 20
```

■ Short interview line

“A higher-order function takes or returns another function—like map, filter, and reduce.”

✅ 5. Explain function scope vs block scope

■ Explanation (simple, conceptual)

Function Scope

- Variables declared with `var` are accessible anywhere inside the function.

Block Scope

- Variables declared with `let` or `const` exist only within `{}` (if, loops, blocks).

■ When/Why is it used

- Function scope → older JS behavior (var), avoid for predictable scoping.
- Block scope → modern JS, cleaner and safer.

■ Example

```
function test() {
  if (true) {
    var a = 10;   // function-scoped
    let b = 20;   // block-scoped
  }
  console.log(a); // 10
  console.log(b); // Error: b is not defined
}
```

■ Short interview line

"var is function-scoped; let and const are block-scoped, existing only inside `{}`."

---

3. Asynchronous JavaScript (Most Important)

✅ 1. What is synchronous vs asynchronous programming?

■ Explanation (simple, conceptual)

Synchronous programming

- Code executes **line by line**
- Next statement waits until the previous one completes

- Blocks the thread

Asynchronous programming

- Code can **start now** and finish **later**
- You don't wait; the program continues executing
- Prevents blocking (very important for browsers & Node.js)

■ When/Why is it used

- Asynchronous needed when doing tasks that take time:
  - API calls
  - Reading files
  - DB queries
  - Timers
  - Heavy computations

■ Example

```
console.log("A");
setTimeout(() => console.log("B"), 1000);
console.log("C");

// Output: A, C, B
```

■ Short interview line

"Synchronous code blocks execution; asynchronous code lets slow tasks run in the background without blocking."

✅ 2. What is the event loop? How does it work?

■ Explanation (simple, conceptual)

The **Event Loop** is the mechanism that enables JavaScript (single-threaded) to handle asynchronous tasks without blocking.

JS has:

1. **Call Stack** – runs code
2. **Web APIs** – timers, fetch, event listeners

3. **Callback / Microtask Queue** – queued callbacks / promises
4. **Event Loop** – **moves tasks from queues to the call stack**

■ When/Why used

- Helps JS run async operations like `setTimeout`, `fetch`, promises
- Makes JS non-blocking despite being single-threaded

■ Example

```
console.log("Start");
setTimeout(() ⇒ console.log("Timeout"), 0);
Promise.resolve().then(() ⇒ console.log("Promise"));
console.log("End");

// Output:
// Start
// End
// Promise     ⟵ microtask
// Timeout     ⟵ callback queue
```

■ Short interview line

"The event loop continuously checks the call stack and queues to run async tasks without blocking."

✅ 3. Explain callbacks, promises, and async/await

■ Explanation (simple, conceptual)

✔ Callback

A function passed to another function and executed later (async).

Problem: leads to *callback hell*.

✔ Promise

An object representing the result of an async operation in the future.

Has states: **pending → fulfilled → rejected**.

✔ async/await

A cleaner way to write asynchronous code using promises under the hood — looks like synchronous code.

■ When/Why used

- Callbacks → simple async tasks
- Promises → better error handling, avoid callback hell
- Async/await → best readability and clean error handling

■ Examples

Callback

```
setTimeout(() => console.log("done"), 1000);
```

Promise

```
fetchData()
  .then(res => console.log(res))
  .catch(err => console.log(err));
```

Async/Await

```
async function getData() {
  try {
    const res = await fetchData();
    console.log(res);
  } catch (e) {
    console.log(e);
  }
}
```

■ Short interview line

“Callbacks are basic async tools, promises avoid callback hell, and async/await makes async code readable.”

✅ 4. What is a promise chain?

■ Explanation (simple, conceptual)

A **promise chain** is when multiple `.then()` calls are linked so the output of one async operation flows into the next.

■ When/Why used

- Running async tasks **in sequence**
- Avoiding nested callbacks
- Clean error handling

■ Example

```
fetchUser()
  .then(user ⇒ fetchOrders(user.id))
  .then(orders ⇒ fetchPayments(orders))
  .then(payments ⇒ console.log(payments))
  .catch(err ⇒ console.log(err));
```

■ Short interview line

"A promise chain lets multiple async operations run sequentially without nested callbacks."

✅ 5. What is `setTimeout()` and how does it work?

■ Explanation (simple, conceptual)

`setTimeout()` schedules a function to run **after a delay**, but it does **not block the thread**.

■ When/Why used

- Delayed execution
- Simulating async behavior
- UI animations, retry logic, debouncing

■ How it works (flow)

1. `setTimeout()` goes to **Web APIs**, timer starts
2. After delay, callback goes to **callback queue**
3. Event loop moves it to the **call stack** when stack is empty
4. Then callback executes

■ Example

```
setTimeout(() => console.log("Hello"), 2000);
```

■ Short interview line

“setTimeout() schedules a callback to run later via Web APIs; it doesn’t block the main thread.”

---

4. DOM & Browser Concepts

✅ 1. What is the DOM?

■ Explanation (simple, conceptual)

DOM (Document Object Model) is a **tree-like representation** of a web page.

JavaScript uses the DOM to **access, modify, or remove** HTML elements dynamically.

■ When/Why is it used

- To update UI (change content, style, attributes)
- To handle user interactions (click, input, scroll)
- To create dynamic applications

■ Example

```
document.getElementById("title").innerText = "Hello JS";
```

■ Short interview line

“DOM is a tree structure of the webpage that JavaScript uses to manipulate UI elements.”

✅ 2. Difference between `document.querySelector` and `getElementById`

■ Explanation (simple)

Method	What it selects	Selector type	Returns
getElementById()	Element by ID	Only ID	Fast, direct reference
querySelector()	First matching element	CSS selectors	More flexible

■ When/Why used

- Use **getElementById** when selecting by unique ID (fastest).
- Use **querySelector** when selecting by class, tag, attributes, or complex selectors.

■ Example

```
document.getElementById("title");
document.querySelector(".card");      // class
document.querySelector("#title");     // id
document.querySelector("div > p");    // CSS selector
```

■ Short interview line

“getElementById is faster but limited to IDs; querySelector is flexible using any CSS selector.”

✅ 3. What is event bubbling vs capturing?

■ Explanation (simple, conceptual)

Events move through the DOM in **two phases**:

✔ Capturing Phase (top → down)

Event travels from **document → parent → target**.

✔ Bubbling Phase (bottom → up)

Event travels from **target → parent → document**.

JavaScript uses **bubbling** by default.

■ When/Why used

- Capturing for rare cases like global event handling
- Bubbling for most UI interactions (clicks, forms)
- Used in **event delegation** (very important)

■ Example

```
element.addEventListener("click", handler, true);   // capturing
element.addEventListener("click", handler, false);  // bubbling
```

■ Short interview line

“Capturing goes top-down; bubbling goes bottom-up. JS uses bubbling by default.”

✅ 4. What is localStorage vs sessionStorage?

■ Explanation (simple)

Feature	localStorage	sessionStorage
Lifetime	Until manually cleared	Clears when tab closes
Capacity	~5–10MB	~5MB
Scope	Shared across tabs (same domain)	Unique per tab
Persistence	Persistent	Temporary

■ When/Why used

- localStorage → user preferences, auth tokens (not sensitive), dark mode settings
- sessionStorage → temporary data like form inputs, session state

■ Example

```
localStorage.setItem("name", "Mohini");
sessionStorage.setItem("count", 5);
```

■ Short interview line

“localStorage persists until cleared; sessionStorage lasts only for the tab session.”

✅ 5. What are events in JavaScript?

■ Explanation (simple, conceptual)

An **event** is an action that occurs in the browser — like a click, scroll, keypress, API completion — and JavaScript can respond to it.

■ When/Why used

- To make websites interactive
- To trigger functions based on user actions or async operations
- Backbone of UI development

■ Example

```
document.getElementById("btn")
  .addEventListener("click", () => console.log("Clicked"));
```

■ Short interview line

“Events are user or system actions (click, input, load) that JavaScript listens to and reacts to.”

---

5. Objects & Prototypes

✅ 1. What is a JavaScript object?

■ Explanation (simple, conceptual)

A **JavaScript object** is a collection of **key–value pairs**.

Keys are strings (or symbols), and values can be anything — number, string, array, function, another object, etc.

JS objects are used to represent real-world entities (user, product, transaction).

■ When/Why is it used

- To store structured data
- To group related properties & behaviors
- Used everywhere in JS (JSON, DOM nodes, classes, arrays)

■ Example

```
const user = {
  name: "Mohini",
  age: 22,
  greet() { console.log("Hello"); }
};
```

■ Short interview line

“An object is a key-value data structure used to represent structured information.”

✅ 2. How to create objects in JS?

■ Explanation (simple)

There are multiple ways to create objects in JavaScript:

✔ Object Literal (most common)

```
const user = { name: "Mohini", age: 22 };
```

✔ Using `new Object()`

```
const user = new Object();
user.name = "Mohini";
```

✔ Constructor function

```
function User(name) {
  this.name = name;
}
const u = new User("Mohini");
```

✔ Class (ES6)

```
class User {
  constructor(name) {
    this.name = name;
  }
}
```

✔ Object.create() — prototypal inheritance

```
const parent = { greet() { console.log("Hello"); } };
const child = Object.create(parent);
```

■ Short interview line

“Objects can be created using literals, constructors, classes, or Object.create for prototype-based inheritance.”

✅ 3. What is prototypal inheritance?

■ Explanation (simple, conceptual)

Prototypal inheritance means:

Objects inherit properties and methods from a prototype object.

Every object in JS has a hidden property `__proto__` that refers to its prototype.

If a property is not found on the object → JS looks up the prototype chain.

■ When/Why is it used

- Code reuse
- Sharing methods across objects
- Behind the scenes of classes in JavaScript

■ Example

```
const parent = { greet: () => console.log("Hello") };
const child = Object.create(parent);

child.greet();  // inherits from parent
```

■ Short interview line

"Prototypal inheritance allows objects to inherit methods from other objects via the prototype chain."

✅ 4. What is the difference between a class and a function constructor?

■ Explanation (simple)

Both are used to create objects, but:

✔ Function Constructor (older way)

- Uses a normal function
- Methods added manually to `prototype`
- More verbose

✔ Class (ES6 syntax)

- Cleaner, modern syntax
- Methods automatically added to prototype
- Supports `extends` for inheritance

- Still uses prototypal inheritance under the hood

■ When/Why used

- Use **class** for modern JS applications
- Use constructor functions only for legacy code compatibility

■ Example

Constructor Function

```
function User(name) {
  this.name = name;
}
User.prototype.greet = function() {
  console.log("Hello");
};
```

Class

```
class User {
  constructor(name) {
    this.name = name;
  }
  greet() {
    console.log("Hello");
  }
}
```

■ Short interview line

"Classes are syntactic sugar over constructor functions, providing cleaner syntax and built-in inheritance."

✅ 5. What is `this` keyword? How does it behave differently?

■ Explanation (simple, conceptual)

`this` refers to the **object that is executing the function**.

But its value changes based on how the function is called.

■ When/Why used

- Accessing object properties
- Object methods
- Constructor functions
- Event handlers
- React components (class-based)

■ Behaviors of `this`

Situation	`this` refers to
Object method	The object
Function by itself	`undefined` in strict mode, `window` in non-strict
Constructor function	New instance created
Arrow function	**Lexical `this` — inherits from parent scope**
Event handler	The DOM element

■ Example

```
const user = {
  name: "Mohini",
  greet() { console.log(this.name); }
};
user.greet(); // "Mohini"
```

Arrow function example

```
const obj = {
  name: "Test",
  show: () => console.log(this.name)  // takes this from parent scope
};
```

■ Short interview line

"`this` refers to the calling object, but arrow functions don't have their own `this` — they use lexical `this`."

---

6. Arrays & Useful Methods

✅ 1. Difference between `map()`, `filter()`, and `reduce()`

■ Explanation (simple, conceptual)

These are higher-order array methods used for transformation, filtering, and aggregation.

Method	Purpose	Returns
map()	Transform each element	**New array** (same length)
filter()	Keep only elements that pass a condition	**New array** (may be shorter)
reduce()	Reduce array to a single value	**Any value** (number, object, array)

■ When/Why is it used

- **map()** → modify data (e.g., convert array of prices to taxes)
- **filter()** → remove unwanted items (e.g., active users only)
- **reduce()** → aggregate results (sum, max, group by)

■ Example

```
const arr = [1, 2, 3, 4];

arr.map(x ⇒ x * 2);       // [2, 4, 6, 8]
arr.filter(x ⇒ x % 2 === 0); // [2, 4]
arr.reduce((sum, x) ⇒ sum + x, 0); // 10
```

■ Short interview line

“map transforms, filter selects, and reduce aggregates values into a single output.”

✅ 2. How does `forEach()` differ from `map()`?

■ Explanation (simple)

Feature	forEach()	map()
Return value	**Nothing (undefined)**	**New transformed array**
Purpose	Loop through items	Transform items
Chainable?	❌ No	✔ Yes
Mutates original?	Can	Should not

■ When/Why is it used

- Use **forEach** → when you just want to loop (side effects like logging or modifying DOM).
- Use **map** → when you want a new transformed array (pure function).

■ Example

```
let arr = [1,2,3];

// forEach
arr.forEach(x => console.log(x));

// map
const doubled = arr.map(x => x * 2);
```

■ Short interview line

"forEach is for looping; map is for returning a new transformed array."

✅ 3. How to remove duplicates from an array?

■ Explanation (simple)

Using a **Set** is the cleanest and most efficient way.

■ When/Why is it used

- Removing duplicate IDs, emails, or values
- Data cleaning
- Preprocessing before analytics/UI rendering

■ Examples

✔ Using Set (best)

```
const arr = [1,2,2,3,4,4];
const unique = [...new Set(arr)];
```

✔ Using filter + indexOf

```
const unique = arr.filter((item, index) => arr.indexOf(item) === index);
```

✔️ Using reduce

```
const unique = arr.reduce((acc, item) =>
  acc.includes(item) ? acc : [...acc, item], []);
```

■ Short interview line

"Use a Set: `unique = [...new Set(arr)]` — simplest and most efficient."

✅ 4. What is array destructuring?

■ Explanation (simple, conceptual)

Destructuring allows you to **extract values** from an array into separate variables in a clean way.

■ When/Why is it used

- Cleaner code
- Easily extract values from API results, function returns, arrays
- Makes variables readable and organized

■ Example

```
const arr = [10, 20, 30];

const [a, b, c] = arr;

console.log(a); // 10
```

Skipping values:

```
const [first, , third] = arr;
```

■ Short interview line

"Array destructuring extracts elements into variables in a simple, readable way."

✅ 5. Example of spread (`...`) and rest operators

■ Explanation (simple)

Both use `...` but serve **opposite purposes**.

✔ Spread Operator — expands an array/object

✔ Rest Operator — collects multiple values into a single variable

■ When/Why is it used

- Spread → copying arrays/objects, merging, passing arguments
- Rest → collecting variable number of parameters

■ Examples

✔ Spread

```
const arr = [1,2,3];
const copy = [...arr];          // copy array
const combined = [...arr, 4,5]; // merge arrays
```

✔ Rest

```
function sum(...nums) {
  return nums.reduce((a, b) => a + b);
}
sum(1,2,3); // 6
```

✔ Spread with objects

```
const obj = { a: 1 };
const newObj = { ...obj, b: 2 };
```

■ Short interview line

“Spread expands arrays/objects; rest collects multiple values into one variable.”

---

7. Let, Const, Arrow Functions (Commonly Asked for JS Fundamentals)

✅ 1. Why can’t you reassign a `const` variable?

■ Explanation (simple, conceptual)

`const` creates a **constant binding** — meaning the **variable reference cannot be changed** after its initial assignment.

This does **not** mean the value is frozen.

For objects & arrays, the reference stays same but internal data can change.

■ When/Why is it used

- For values that should **never change** (API keys, config values).
- Makes code predictable & avoids accidental reassignments.

■ Example

```
const x = 10;
x = 20;  // ❌ Error: Assignment to constant variable

const obj = { name: "Mohini" };
obj.name = "Updated"; // ✔ Allowed (reference same)
```

■ Short interview line

“const prevents reassigning the variable reference, though object values inside can still change.”

✅ 2. Why arrow functions don’t have their own `this` ?

■ Explanation (simple, conceptual)

Arrow functions use **lexical** `this`, meaning:

👉 They **inherit** `this` **from the surrounding scope**,

instead of creating their own `this` like regular functions.

That's why arrow functions are great for callbacks but cannot be used as constructors.

■ When/Why is it used

- When you want to avoid `this` confusion
- Inside event handlers, array methods, and callbacks

- Prevents needing `.bind(this)`

■ Example

```
const obj = {
  name: "Mohini",
  normal() { console.log(this.name); },
  arrow: () => console.log(this.name)
};

obj.normal(); // "Mohini"
obj.arrow();  // undefined (inherits from global)
```

■ Short interview line

"Arrow functions inherit `this` from their parent scope; they don't create their own `this`."

✅ 3. What is the temporal dead zone (TDZ)?

■ Explanation (simple, conceptual)

The **Temporal Dead Zone** is the period between a variable's **hoisting** and its **actual initialization** where it **cannot be accessed**.

It applies to variables declared with **let** and **const**.

■ When/Why is it used

- Ensures safer behavior than `var`
- Prevents accidental access before initialization
- Helps catch logical coding mistakes

■ Example

```
console.log(a); // ❌ ReferenceError (TDZ)
let a = 10;
```

The variable `a` is hoisted but not initialized → accessing it before its declaration throws an error.

■ Short interview line

"The Temporal Dead Zone is the period where `let` and `const` exist but cannot be accessed before initialization."

---

8. JSON

✅ 1. What is JSON?

■ Explanation (simple, conceptual)

JSON (JavaScript Object Notation) is a lightweight data format used to store and transmit data between a client and server.

It is text-based, language-independent, and easy for both humans and machines to read.

■ When/Why is it used

- To exchange data over APIs (REST APIs commonly return JSON)
- To store configuration, metadata, and structured information
- To represent objects in a simple, standard format

■ Example

```
{
  "name": "Mohini",
  "age": 22,
  "isActive": true
}
```

■ Short interview line

"JSON is a lightweight data-interchange format used for client–server communication."

✅ 2. JSON.parse() vs JSON.stringify()

■ Explanation (simple)

Method	Purpose
JSON.parse()	Converts **JSON string → JavaScript object**
JSON.stringify()	Converts **JavaScript object → JSON string**

■ When/Why is it used

- **parse()** → when receiving JSON from backend (convert string to object)

- **stringify()** → when sending data to backend or storing in localStorage

■ Examples

✔ JSON.parse()

```
const jsonStr = '{"name": "Mohini", "age": 22}';
const obj = JSON.parse(jsonStr);
// obj = { name: "Mohini", age: 22 }
```

✔ JSON.stringify()

```
const obj = { name: "Mohini", age: 22 };
const jsonStr = JSON.stringify(obj);
// jsonStr = '{"name":"Mohini","age":22}'
```

■ Short interview line

"JSON.parse converts JSON to object; JSON.stringify converts object to JSON."

✅ 3. How do you send JSON data to a backend API?

■ Explanation (simple, conceptual)

You send JSON using **HTTP POST/PUT requests** with the `Content-Type: application/json` header.

JavaScript converts the object to JSON using `JSON.stringify()`.

■ When/Why is it used

- To send form data
- To update resources
- To create new records (e.g., creating an expense in your Expense Tracker)

■ Example / Code snippet

Using `fetch()`:

```
fetch("https://api.example.com/expenses", {
  method: "POST",
  headers: {
```

```
    "Content-Type": "application/json"
  },
  body: JSON.stringify({
    userId: 1,
    amount: 500,
    category: "Food"
  })
})
.then(res => res.json())
.then(data => console.log(data))
.catch(err => console.error(err));
```

■ Short interview line

“We send JSON using fetch/AJAX with Content-Type: application/json and JSON.stringify() in the request body.”

---

9. Error Handling

✅ 1. How do you handle errors in JavaScript?

■ Explanation (simple, conceptual)

JavaScript handles errors using:

- **try...catch** (synchronous errors)
- **Promises** `.catch()` (asynchronous errors)
- **async/await with try...catch**
- **Error objects** (`throw new Error()`)
- **Global handlers** (fallback):
  - `window.onerror`
  - `unhandledrejection`

■ When/Why is it used

- To prevent the application from crashing
- To show meaningful error messages to users
- To gracefully recover from API failures, invalid inputs, or runtime errors

■ Example

```
try {
  const data = JSON.parse("invalid json");
} catch (err) {
  console.log("Error occurred:", err.message);
}
```

■ Short interview line

“I handle errors using try/catch for sync code and `.catch()` or async/await patterns for async errors.”

✅ 2. What is try...catch?

■ Explanation (simple, conceptual)

`try...catch` is used to handle **runtime errors** without stopping the entire program.

The `try` block runs the code; if it fails, execution jumps to the `catch` block.

■ When/Why is it used

- To handle invalid operations like parsing JSON
- To catch unexpected runtime errors
- To prevent the UI or Node server from crashing

■ Example

```
try {
  let result = riskyOperation();
  console.log(result);
} catch (error) {
  console.log("Something went wrong:", error.message);
}
```

■ Short interview line

“try/catch runs code safely and catches runtime errors without crashing the app.”

✅ 3. What happens if you don’t handle a promise rejection?

■ Explanation (simple, conceptual)

If a promise rejects and there is **no** `.catch()`, it becomes an **Unhandled Promise Rejection**.

In browsers → shows a warning.

In Node.js → may **crash the process** (default behavior in strict modes).

■ When/Why is it important

- API failures
- Network issues
- Incorrect promise logic

  If `.catch()` is missing, bugs become difficult to trace and can break the application.

■ Example

```
Promise.reject("Error!");
// No catch → Unhandled Promise Rejection
```

Handled version:

```
fetchData()
  .then(res => console.log(res))
  .catch(err => console.log("Handled:", err));
```

■ Short interview line

"Unhandled promise rejections cause warnings in browser and can crash Node.js — always use `.catch()` or async/await try/catch."

---

10. Node.js

✅ 1. What is Node.js?

■ Explanation (simple, conceptual)

Node.js is a **JavaScript runtime** built on Chrome's V8 engine that allows JavaScript to run **outside the browser**, mainly for backend development.

■ When/Why is it used

- Build REST APIs
- Real-time apps (chat, notifications)
- Microservices
- Streaming, event-based applications
- High concurrency, non-blocking architecture

■ Example

```
const http = require("http");
http.createServer((req, res) => {
  res.end("Hello Node");
}).listen(3000);
```

■ Short interview line

“Node.js lets us run JavaScript on the server using a fast, non-blocking, event-driven architecture.”

✅ 2. What is NPM?

■ Explanation (simple, conceptual)

NPM (**Node Package Manager**) is the default package manager for Node.js.

It stores reusable libraries and helps manage project dependencies.

■ When/Why is it used

- Install packages (Express, React, Lodash, bcrypt)
- Manage versions
- Run scripts (`npm start`, `npm test`)

■ Example

```
npm install express
```

■ Short interview line

“NPM is the package manager that installs and manages dependencies for Node.js applications.”

✅ 3. Difference between CommonJS and ES Modules

■ Explanation (simple)

Feature	CommonJS	ES Modules (ESM)
Syntax	`require()` / `module.exports`	`import` / `export`
Loading	Synchronous	Asynchronous
Default in	Node.js older versions	Modern JS + Node (ESM support)
Behavior	Dynamic loading	Static structure (better optimization)

■ When/Why used

- **CommonJS** → traditional Node.js projects
- **ESM** → modern JS apps, bundlers, client-side JS, cleaner syntax

■ Examples

CommonJS

```
const express = require("express");
module.exports = app;
```

ES Modules

```
import express from "express";
export default app;
```

■ Short interview line

"CommonJS uses require(), while ES Modules use import/export with static, modern syntax."

✅ 4. What is middleware in Node.js?

■ Explanation (simple, conceptual)

Middleware is a **function that runs between the request and the response** in a Node.js/Express app.

It can modify:

- request

- response
- flow of the application (next())

■ When/Why is it used

- Logging
- Authentication
- Validation
- Parsing JSON
- Error handling

■ Example

```
app.use((req, res, next) => {
  console.log("Request received");
  next();  // move to next middleware
});
```

■ Short interview line

“Middleware is a function in Express that processes requests before sending a response.”

✅ 5. What is the event-driven architecture in Node?

■ Explanation (simple, conceptual)

Node.js uses an **event-driven, non-blocking** architecture.

Instead of waiting for operations to finish (like file read, DB query), Node registers **callbacks** and continues execution.

The **event loop** handles incoming events and executes callbacks asynchronously.

■ When/Why used

- High concurrency
- Real-time systems (chats, notifications, fraud alerts)
- Handling many simultaneous requests
- Preventing blocking operations

■ Example

```
const EventEmitter = require("events");
const emitter = new EventEmitter();

emitter.on("greet", () => console.log("Hello"));
emitter.emit("greet");
```

■ Short interview line

“Node’s event-driven architecture lets it handle thousands of requests using a single thread through callbacks and the event loop.”

⭐ Scenario-Based Questions

1. How did you use JavaScript in your Expense Tracker analytics dashboard?

■ Explanation (simple, conceptual)

I used JavaScript to fetch backend data, transform it into chart-ready series, update the DOM/reactive components, and add interactive behaviors (filters, drill-downs, date pickers). JS acted as the glue: calling APIs, formatting results, and feeding chart libraries.

■ When/Why is it used

- When users need real-time/near-real-time visuals (budgets, trends).
- To provide interactive UX: filter by date/category, hover tooltips, and export CSV.
- To offload simple aggregations or formatting from backend for faster UI responsiveness.

■ Example / Code snippet

```
// fetch + transform + feed chart
async function loadMonthlyChart(userId, start, end) {
  try {
    const res = await fetch(`/api/expenses?userId=${userId}&from=${start}&to=${en
d}`);
    const data = await res.json(); // [{category, day, total}, ...]
    const series = transformToChartSeries(data); // group by category
```

```
    chart.update({ series }); // chart is a charting lib instance
  } catch (err) {
    showToast("Unable to load chart. Try again.");
  }
}
```

■ Short summary line to speak in interview

“I used JS to fetch, transform and feed chart libraries so dashboards update interactively and quickly for users.”

---

2. How did you validate inputs using JS on the frontend?

■ Explanation (simple, conceptual)

Frontend validation used a layered approach: immediate client-side checks for UX (required fields, numeric ranges, date format), followed by a secondary validation before sending to backend. This improves UX and reduces bad requests.

■ When/Why is it used

- Prevent obvious errors (empty amount, invalid date) before network calls.
- Provide instant feedback (inline error messages) and reduce server load.
- Still rely on server-side validation for security and authoritative checks.

■ Example / Code snippet

```
function validateExpense({ amount, category, date }) {
  const errors = {};
  if (!amount || isNaN(amount) || +amount <= 0) errors.amount = "Enter a valid amou
nt";
  if (!category) errors.category = "Category required";
  if (isNaN(Date.parse(date))) errors.date = "Invalid date";
  return errors;
}

form.addEventListener("submit", e => {
  e.preventDefault();
  const payload = getFormData();
  const errors = validateExpense(payload);
  if (Object.keys(errors).length) showFormErrors(errors);
  else submitExpense(payload);
});
```

■ Short summary line to speak in interview

“I implemented client-side validation for instant UX feedback and always enforced server-side checks for security.”

---

3. How did you handle async operations such as fetching expense data from backend?

■ Explanation (simple, conceptual)

I used `async/await` for readability, layered with error handling and cancellation/support for race conditions (e.g., only use latest response). For repeated calls (filters, typeahead) I debounced requests and used caching for hot data (short TTL).

■ When/Why is it used

- To keep UI responsive while waiting for network results.
- To avoid race conditions when users change filters quickly.
- To reduce server load with debounce and caching.

■ Example / Code snippet

```
let currentController = null;

async function fetchExpenses(params) {
  if (currentController) currentController.abort();
  currentController = new AbortController();

  try {
    const res = await fetch(`/api/expenses?${qs(params)}`, { signal: currentControlle
r.signal });
    return await res.json();
  } catch (err) {
    if (err.name === 'AbortError') return; // ignore cancelled request
    throw err; // bubble up
  } finally {
    currentController = null;
  }
}

// Debounce wrapper example
const debouncedFetch = debounce((params) => fetchExpenses(params).then(rende
r), 300);
```

■ Short summary line to speak in interview

“I used async/await with AbortController, debounce and caching to make async fetches reliable, cancelable and efficient.”

---

4. How did you optimize data rendering on dashboard charts?

■ Explanation (simple, conceptual)

I optimized rendering by minimizing DOM updates, using virtualized lists for long tables, sending pre-aggregated data from the backend or materialized views, and feeding charts with only the necessary series/points (sampling or downsampling for long time ranges).

■ When/Why is it used

- For large datasets (months/years of transactions) to keep charts responsive.
- To reduce CPU and repaint costs in the browser and lower payload sizes.

■ Example / Code snippet

```
// 1) Request aggregated data for UI
// Backend: /api/expenses/summary?period=monthly

// 2) Client-side: update only changed series
function updateChart(chart, newSeries) {
  chart.series.forEach((s, i) => s.update(newSeries[i], false));
  chart.redraw();
}

// 3) Virtualize long tables (conceptual)
<VirtualList items={rows} rowHeight={40} renderRow={RowComponent} />
```

■ Short summary line to speak in interview

“I reduced rendering work by using pre-aggregated data, incremental chart updates, and virtualization for large tables so dashboards stay snappy.”

---

5. If an API call fails (e.g., fetching fraud detection result), how will you show error to user?

■ Explanation (simple, conceptual)

User-facing error handling includes: friendly inline errors (toast/banner), retry options, lightweight fallback UI, and logging the error for debugging. Errors are categorized (network, auth, server) to show appropriate messages and actions.

■ When/Why is it used

- To inform users and provide clear next steps (retry, contact support).
- To avoid confusing raw error dumps and improve trust.
- To ensure operations are idempotent and safe on retry.

■ Example / Code snippet

```
async function loadFraudResult(txnId) {
  try {
    showLoader();
    const res = await fetch(`/api/fraud/${txnId}`);
    if (!res.ok) throw new Error(await res.text());
    const data = await res.json();
    renderFraud(data);
  } catch (err) {
    // categorize
    if (err.name === 'TypeError') showBanner("Network error — check your connecti
on.");
    else showBanner("Unable to fetch fraud result. Retry?");
    logger.error("Fraud API error", { txnId, err });
    showRetryButton(() => loadFraudResult(txnId));
  } finally {
    hideLoader();
  }
}
```

■ Short summary line to speak in interview

“I show friendly inline messages (banner/toast), provide retry/fallbacks, and log errors for debugging so users know what to do.”

---

6. How did you structure your Node.js code while working with REST APIs?

■ Explanation (simple, conceptual)

I followed a modular, layered architecture: **routes → controllers → services → repositories/models** plus middleware for auth, validation, and error handling. This

separation keeps controllers thin, business logic in services, and DB access isolated, making the code testable and maintainable.

■ When/Why is it used

- To improve readability, testability, and reusability.
- To enable CI/CD, easier code reviews, and independent scaling of services.
- To allow swapping DB or external integrations with minimal changes.

■ Example / Code snippet (structure)

```
/src
  /routes
    expenses.routes.js     // route definitions
  /controllers
    expenses.controller.js // parse input, call service, send response
  /services
    expenses.service.js    // business logic, transactions
  /repositories
    expenses.repo.js       // raw DB queries or ORM
  /middleware
    auth.js, validate.js, errorHandler.js
  app.js                   // express setup, middleware, routes
```

```
// controller (thin)
async function createExpense(req, res, next) {
  try {
    const result = await expenseService.createExpense(req.user.id, req.body);
    res.status(201).json(result);
  } catch (err) { next(err); }
}
```

```
// service (logic + transaction)
async function createExpense(userId, payload) {
  return db.transaction(async (tx) => {
    await repo.insertExpense(tx, { userId, ...payload });
    await analytics.updateUserDaily(tx, userId, payload.amount);
    return { success: true };
  });
}
```

■ Short summary line to speak in interview

“I organized Node code into routes/controllers/services/repositories with middleware and transactional services to keep logic testable, maintainable and robust.”
