## ✅ Now Let’s Tackle All the JavaScript Advanced Concepts You Asked

## 🔹 1. let, const, var — Scope & Hoisting

| Feature | var | let / const |
| --- | --- | --- |
| Scope | Function-scoped | Block-scoped |
| Hoisting | Yes, value is undefined | Yes, but **Temporal Dead Zone** |
| Redeclaration | Allowed | ❌ Not allowed (in same scope) |
| Use case | Legacy | Modern JS (always prefer const, use let only when you reassign) |

## 🔹 2. Closures

A **closure** is when a function "remembers" variables from its outer lexical scope even after the outer function has returned.

js

CopyEdit

function outer() {

let counter = 0;

return function inner() {

counter++;

console.log(counter);

};

}

const fn = outer(); // returns inner

fn(); // 1

fn(); // 2

Here, inner() **closes over** counter.

## 🔹 3. Promises — Manual Creation & Chaining

js

CopyEdit

function delay(ms) {

return new Promise((resolve) => {

setTimeout(() => resolve(`Waited ${ms}ms`), ms);

});

}

delay(1000)

.then(result => {

console.log(result);

return delay(500);

})

.then(result => console.log(result));

Promises are for handling **async logic** without callback hell.

## 🔹 4. async/await vs .then()

| Feature | .then() | async/await |
| --- | --- | --- |
| Syntax | Chained | Linear, cleaner |
| Error handling | .catch() | try/catch |
| Readability | Gets messy with nesting | Cleaner flow |
| Use case | When chaining logic | For readable, top-down code |

js

CopyEdit

async function run() {

try {

const res = await delay(1000);

console.log(res);

} catch (e) {

console.error(e);

}

}

## 🔹 5. Event Loop: Microtask vs Macrotask

| Task Type | Examples |
| --- | --- |
| **Microtask** | Promise.then, queueMicrotask |
| **Macrotask** | setTimeout, setInterval, DOM events |

🔁 Event Loop Priority:

bash

CopyEdit

Run call stack ➜ Run all microtasks ➜ Run one macrotask ➜ repeat

js

CopyEdit

setTimeout(() => console.log("timeout"), 0);

Promise.resolve().then(() => console.log("promise"));

// Output:

// promise

// timeout

## 🔹 6. String Interning

js

CopyEdit

const a = "hello";

const b = "hello";

console.log(a === b); // true — same memory (interned)

const c = new String("hello");

console.log(a === c); // false — object vs primitive

JS **automatically reuses** string literals for performance — that’s **interning**.

## 🔹 7. Prototypes & Inheritance

Every JS object inherits from its prototype:

js

CopyEdit

function User(name) {

this.name = name;

}

User.prototype.sayHi = function() {

return `Hi, I'm ${this.name}`;

};

const u = new User("Vishnu");

console.log(u.sayHi()); // works via prototype

JavaScript uses **prototype chaining** (not classes internally like Java/Python).

## 🔹 8. this — Regular vs Arrow Functions

| Case | Behavior |
| --- | --- |
| Normal function | this = caller |
| Arrow function | this = lexical parent |

js

CopyEdit

const obj = {

name: "Vishnu",

greet: function() {

return () => console.log(this.name);

}

};

obj.greet()(); // Vishnu — arrow keeps outer `this`

## 🔹 9. Shallow vs Deep Copy

js

CopyEdit

// Shallow copy

const a = { x: 1, nested: { y: 2 } };

const b = { ...a };

b.nested.y = 99; // a.nested.y is now 99 — shared ref!

// Deep copy (safe)

const deep = structuredClone(a);

Or use JSON.parse(JSON.stringify(obj)) — but it loses functions/dates.

## 🔹 10. Spread vs Rest Operator

js

CopyEdit

// Spread: expands

const arr = [1, 2];

const newArr = [...arr, 3]; // [1, 2, 3]

// Rest: collects

function sum(...args) {

return args.reduce((a, b) => a + b);

}

sum(1, 2, 3); // 6

Same ... operator — but **spread = expand**, **rest = collect**.

## 🔹 11. Debounce vs Throttle

| Concept | Use Case | Behavior |
| --- | --- | --- |
| **Debounce** | Search box, resize | Waits for a pause to trigger |
| **Throttle** | Scroll, mousemove | Ensures fixed rate (once per X ms) |

### Debounce (trigger after inactivity):

js

CopyEdit

function debounce(fn, delay) {

let timer;

return (...args) => {

clearTimeout(timer);

timer = setTimeout(() => fn(...args), delay);

};

}

### Throttle (limit frequency):

js

CopyEdit

function throttle(fn, limit) {

let lastCall = 0;

return (...args) => {

const now = Date.now();

if (now - lastCall >= limit) {

lastCall = now;

fn(...args);

}

};

}

## 🎯 TL;DR: What You Should Remember for Interview

| Concept | Quick Summary |
| --- | --- |
| let/const/var | Use const by default, let if reassign needed |
| Closures | Functions “remember” their outer scope |
| Promises | Use .then / .catch, or better: async/await |
| Error Wrapper | Use asyncHandler(fn) to catch async errors in Express |
| Event Loop | Microtasks run before macrotasks |
| Interning | Literal strings are interned (same memory) |
| this | Arrow retains outer this, normal depends on caller |
| Prototypes | JS inheritance is via prototype chain |
| Spread/Rest | ... = expands or collects depending on context |
| Debounce/Throttle | Use to control frequent frontend/backend events |

✅ 1. JavaScript Data Types

### 🔹 ****Primitive Data Types****

These are immutable and stored by value:

* string → "hello"
* number → 42, 3.14, -0
* boolean → true, false
* null → intentional absence of value
* undefined → variable declared but not assigned
* bigint → 1234567890123456789012345678901234567890n
* symbol → Symbol("id") (unique and immutable)

js

CopyEdit

let name = "Vishnu"; // string

let age = 25; // number

let hasJob = true; // boolean

let x = undefined;

let y = null;

### 🔹 ****Non-Primitive Data Types (Reference Types)****

Stored by reference:

* Object
* Array
* Function
* Date, RegExp, Map, Set, etc.

js

CopyEdit

let person = { name: "Vishnu", age: 25 };

let arr = [1, 2, 3];

let greet = function() { console.log("Hello!"); };

## ✅ 2. Variable Declaration

| Type | Re-assignable | Block Scoped | Hoisted |
| --- | --- | --- | --- |
| var | ✅ | ❌ (function) | ✅ |
| let | ✅ | ✅ | ❌ |
| const | ❌ (but object props mutable) | ✅ | ❌ |

js

CopyEdit

const user = { name: "Vishnu" };

user.name = "Kurup"; // ✅

## ✅ 3. Conditionals

js

CopyEdit

if (age >= 18) {

console.log("Adult");

} else if (age > 12) {

console.log("Teen");

} else {

console.log("Child");

}

// Ternary

let msg = age >= 18 ? "Adult" : "Minor";

## ✅ 4. Loops

### 🔹 for loop

js

CopyEdit

for (let i = 0; i < 5; i++) {

console.log(i);

}

### 🔹 while loop

js

CopyEdit

let i = 0;

while (i < 5) {

console.log(i);

i++;

}

### 🔹 do...while

js

CopyEdit

let i = 0;

do {

console.log(i);

i++;

} while (i < 5);

### 🔹 for...of (iterates values)

js

CopyEdit

for (let num of [1, 2, 3]) {

console.log(num);

}

### 🔹 for...in (iterates keys)

js

CopyEdit

let obj = { a: 1, b: 2 };

for (let key in obj) {

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

}

## ✅ 5. Operators

### 🔹 Arithmetic

+, -, \*, /, %, \*\*

### 🔹 Comparison

== vs === → === is strict equality (no type coercion)

js

CopyEdit

'5' == 5 // true

'5' === 5 // false

### 🔹 Logical

* && → and
* || → or
* ! → not

js

CopyEdit

true && false // false

true || false // true

!false // true

## ✅ 6. Type Coercion

JavaScript is loosely typed. So:

js

CopyEdit

'5' + 1 // '51' (string)

'5' - 1 // 4 (number)

null == undefined // true

null === undefined // false

## ✅ 7. Arrays and Objects

js

CopyEdit

let fruits = ["apple", "banana"];

let person = { name: "Vishnu", age: 25 };

// Access

fruits[0]; // 'apple'

person.name; // 'Vishnu'

## ✅ 8. Functions

### Function Declaration

js

CopyEdit

function greet(name) {

return `Hello ${name}`;

}

### Arrow Function

js

CopyEdit

const greet = (name) => `Hello ${name}`;

## ✅ 9. typeof and instanceof

js

CopyEdit

typeof "hello" // string

typeof 5 // number

typeof null // object (quirk)

typeof undefined // undefined

typeof [] // object

[] instanceof Array // true

{} instanceof Object // true

## ✅ 10. Hoisting

* var and function declarations are hoisted.
* let and const are hoisted **but not initialized** (TDZ - Temporal Dead Zone).

js

CopyEdit

console.log(a); // undefined

var a = 5;

console.log(b); // ReferenceError

let b = 10;

## ✅ 11. == vs === Table

| Expression | == | === |
| --- | --- | --- |
| '5' == 5 | true | false |
| null == undefined | true | false |
| 0 == false | true | false |

## ✅ 1. ****Closures****

**Definition**: A closure is when an inner function remembers and can access variables from its outer function, even after the outer function has returned.

js

CopyEdit

function outer() {

let count = 0;

return function inner() {

count++;

console.log(count);

};

}

const counter = outer();

counter(); // 1

counter(); // 2 → Still remembers `count`

**Why used?**

* Data privacy
* Function factories
* In async loops (e.g. setTimeout)

**Interview Q**: How do closures help emulate private variables in JS?

## ✅ 2. ****Scope & Lexical Environment****

**Scope**: Determines the accessibility of variables.

* **Global Scope**
* **Function Scope**
* **Block Scope (**let**/**const**)**

**Lexical Scope**: Scope is determined by the position of code during **write time**, not run time.

js

CopyEdit

function a() {

let x = 2;

function b() {

console.log(x); // b has access to x due to lexical scope

}

b();

}

## ✅ 3. ****Prototypes & Inheritance****

JS is **prototype-based** (not class-based originally).

js

CopyEdit

function Person(name) {

this.name = name;

}

Person.prototype.sayHi = function() {

return `Hi, I’m ${this.name}`;

};

const user = new Person("Vishnu");

user.sayHi(); // Works due to prototype inheritance

Modern class syntax is **syntactic sugar** over this.

js

CopyEdit

class Person {

constructor(name) {

this.name = name;

}

sayHi() {

return `Hi, I’m ${this.name}`;

}

}

**Interview Q**: What’s the difference between prototype chain and class inheritance in JS?

## ✅ 4. ****Event Loop, Call Stack, and Microtasks****

JavaScript is **single-threaded**, but asynchronous using:

* **Call Stack**
* **Web APIs**
* **Callback Queue (macrotasks)**
* **Microtask Queue** (Promises)

js

CopyEdit

console.log("Start");

setTimeout(() => console.log("Timeout"), 0);

Promise.resolve().then(() => console.log("Promise"));

console.log("End");

// Output:

// Start

// End

// Promise

// Timeout

**Interview Q**: Why does Promise log before setTimeout?

## ✅ 5. ****async/await & Promises****

### 🔹 Promises

js

CopyEdit

const promise = new Promise((resolve, reject) => {

setTimeout(() => resolve("Done!"), 1000);

});

promise.then(console.log); // "Done!"

### 🔹 async/await

js

CopyEdit

async function fetchData() {

try {

const result = await promise;

console.log(result);

} catch (e) {

console.error(e);

}

}

**Interview Q**: How is error handling done in async/await vs then/catch?

## ✅ 6. this ****Keyword****

* In global scope: this === window (in browser)
* In function: Depends on how it's **called**
* In arrow function: this is **lexically bound**

### 🔹 Example

js

CopyEdit

const obj = {

name: "Vishnu",

greet: function () {

return `Hello, ${this.name}`;

},

};

obj.greet(); // "Hello, Vishnu"

const greet = obj.greet;

greet(); // "Hello, undefined" → because `this` lost

**Fix:** Use bind(), arrow functions, or class fields

## ✅ 7. ****ES6+ Features****

### 🔹 Destructuring

js

CopyEdit

const user = { name: "Vishnu", age: 25 };

const { name, age } = user;

### 🔹 Spread/Rest

js

CopyEdit

const arr = [1, 2, 3];

const newArr = [...arr, 4];

function sum(...nums) {

return nums.reduce((a, b) => a + b);

}

### 🔹 Optional Chaining

js

CopyEdit

let obj = { user: { profile: null } };

console.log(obj.user?.profile?.name); // undefined, not error

### 🔹 Nullish Coalescing

js

CopyEdit

let val = 0;

console.log(val ?? 42); // 0, not 42

## ✅ 8. ****Currying****

Breaking a function into **a chain of functions**, each taking a single argument.

js

CopyEdit

function add(a) {

return function(b) {

return function(c) {

return a + b + c;

};

};

}

add(1)(2)(3); // 6

**Interview Q**: What are benefits of currying?  
✅ Reusability, partial function application

## 🔥 Recap: You’re Now Interview-Ready In

| Topic | Interview Worthy Questions |
| --- | --- |
| Closures | Data privacy, async fixes |
| Scope | Lexical vs runtime scope |
| Prototypes | Difference with classes |
| Event Loop | Microtask vs macrotask |
| Promises | Async flow, chaining |
| this | Arrow vs regular |
| ES6+ | Must-know for modern dev |
| Currying | Functional programming & reuse |

✅ Code First

js

CopyEdit

console.log("Start");

setTimeout(() => {

console.log("Timeout");

Promise.resolve().then(() => console.log("Inner Promise"));

}, 0);

Promise.resolve().then(() => console.log("Outer Promise"));

console.log("End");

## ✅ The Concept: JavaScript Execution Engine

🧠 What powers this?

* **Call Stack**: Executes code line-by-line.
* **Web APIs**: Handles setTimeout, DOM events, etc.
* **Callback Queue** (Macrotask Queue): For setTimeout, setInterval, etc.
* **Microtask Queue**: For Promise.then, MutationObserver, queueMicrotask.

### ⚙️ Order of Priority

Call Stack → Microtask Queue → Macrotask Queue

## ✅ Step-by-Step Execution (Line-by-Line)

### ⏱ ****Phase 1: Global Code Execution (Call Stack)****

1. console.log("Start")  
   ➡️ Goes on call stack → prints → removed from stack
2. setTimeout(...)  
   ➡️ Moved to **Web API** (does NOT block anything)  
   ➡️ After 0ms, callback is scheduled in **Macrotask Queue**
3. Promise.resolve().then(...)  
   ➡️ .then(...) is pushed to **Microtask Queue**
4. console.log("End")  
   ➡️ Goes on stack → prints → removed

🧾 So far, Console Output:

sql

CopyEdit

Start

End

### 🌀 ****Phase 2: Microtask Queue (High Priority)****

Before macrotasks are touched, JS drains the **microtask queue**:

* Promise.resolve().then(...) → prints:

javascript

CopyEdit

Outer Promise

### 📦 ****Phase 3: Macrotask Queue (Now handle setTimeout)****

Now that stack is empty and microtasks are done, JS takes the next **macrotask**:

js

CopyEdit

setTimeout(() => {

console.log("Timeout");

Promise.resolve().then(() => console.log("Inner Promise"));

}, 0);

* Pushes the setTimeout callback on the call stack
* Executes console.log("Timeout") → prints:

nginx

CopyEdit

Timeout

* Then sees a Promise.then(...)  
  ➡️ Adds console.log("Inner Promise") to **Microtask Queue**

### 🔄 Back to Microtasks Again (after setTimeout finishes)

* Drain Microtask Queue:
  + console.log("Inner Promise") → prints:

javascript

CopyEdit

Inner Promise

## ✅ Final Console Output:

sql

CopyEdit

Start

End

Outer Promise

Timeout

Inner Promise

## 🔁 Visual Execution Table

| Phase | What Happens | Output |
| --- | --- | --- |
| Global Code | console.log("Start") | Start |
|  | setTimeout(...) → Web API → Macrotask |  |
|  | Promise.resolve().then(...) → Microtask |  |
|  | console.log("End") | End |
| Microtasks | Outer Promise | Outer Promise |
| Macrotasks | setTimeout() callback runs | Timeout |
|  | Inside: Promise.then(...) → Microtask |  |
| Microtasks | Inner Promise | Inner Promise |

## 🔥 Interview Insight: Nested Event Loops

Imagine inside a setInterval, you're doing async stuff:

js

CopyEdit

setInterval(() => {

console.log("Tick");

Promise.resolve().then(() => console.log("Microtask inside Interval"));

setTimeout(() => console.log("Another Macrotask"), 0);

}, 3000);

Every 3s:

1. Macrotask (setInterval) executes → prints Tick
2. Adds Promise’s .then(...) to Microtask Queue
3. Adds new setTimeout(...) to Macrotask Queue
4. Executes Microtasks → Microtask inside Interval
5. Then later → Another Macrotask

## ⚠️ Summary: Golden Rules

✅ JS engine **always empties microtask queue** after each task (render, macrotask, etc)  
✅ **setTimeout/setInterval → macrotask**  
✅ **Promises → microtask**, so they have **higher priority**  
✅ Inside a macrotask, if you queue another microtask (e.g. Promise.then), it runs **before any new macrotask**

# ✅ JavaScript Built-in Methods (by Data Type)

## 🔢 1. Number Methods

js

CopyEdit

let num = 123.456;

num.toFixed(2); // "123.46"

num.toString(); // "123.456"

Number.isInteger(num); // false

Math.round(num); // 123

Math.floor(num); // 123

Math.ceil(num); // 124

Math.abs(-42); // 42

Math.sqrt(25); // 5

Math.pow(2, 3); // 8

parseInt("42"); // 42

parseFloat("42.5"); // 42.5

## 🧵 2. String Methods

js

CopyEdit

let str = " Hello World ";

str.length; // 13

str.trim(); // "Hello World"

str.toUpperCase(); // " HELLO WORLD "

str.toLowerCase(); // " hello world "

str.includes("World"); // true

str.startsWith(" Hello"); // true

str.endsWith(" "); // true

str.indexOf("o"); // 5

str.lastIndexOf("o"); // 8

str.replace("World", "JS"); // " Hello JS "

str.slice(1, 6); // "Hello"

str.split(" "); // ["", "Hello", "World", ""]

str.repeat(3); // " Hello World Hello World Hello World "

## 📦 3. Array Methods

js

CopyEdit

let arr = [1, 2, 3, 4, 5];

arr.length; // 5

arr.push(6); // adds to end

arr.pop(); // removes from end

arr.unshift(0); // adds to start

arr.shift(); // removes from start

arr.includes(3); // true

arr.indexOf(2); // 1

arr.slice(1, 3); // [2, 3]

arr.splice(2, 1); // remove 1 element at index 2

arr.concat([7, 8]); // [1, 2, 4, 5, 7, 8]

arr.join("-"); // "1-2-4-5"

arr.map(x => x \* 2); // [2, 4, 8, 10]

arr.filter(x => x > 3); // [4, 5]

arr.reduce((a, b) => a + b, 0); // 12

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

arr.find(x => x > 3); // 4

arr.every(x => x < 6); // true

arr.some(x => x === 2); // true

## 🧱 4. Object Methods

js

CopyEdit

let obj = { name: "Vishnu", age: 24 };

Object.keys(obj); // ["name", "age"]

Object.values(obj); // ["Vishnu", 24]

Object.entries(obj); // [["name", "Vishnu"], ["age", 24]]

Object.hasOwn(obj, "name"); // true

Object.assign({}, obj); // clone

## 🧠 5. Type Conversion

js

CopyEdit

String(123); // "123"

Number("123"); // 123

Boolean(0); // false

parseInt("42px"); // 42

parseFloat("3.14"); // 3.14

123..toString(); // "123"

## 📆 6. Date Methods

js

CopyEdit

let now = new Date();

now.toString(); // "Wed Jun 19 2025 ..."

now.getFullYear(); // 2025

now.getMonth(); // 5 (June)

now.getDate(); // 19

now.getDay(); // 4 (Thursday)

now.getHours(); // current hour

now.toISOString(); // "2025-06-19T..."

Date.now(); // timestamp in ms

## 🎛️ 7. JSON Methods

js

CopyEdit

let obj = { name: "Vishnu", age: 24 };

let jsonStr = JSON.stringify(obj); // '{"name":"Vishnu","age":24}'

JSON.parse(jsonStr); // { name: "Vishnu", age: 24 }

## 🧪 8. Common Utility Methods

js

CopyEdit

typeof 123; // "number"

typeof "hello"; // "string"

Array.isArray([1]); // true

isNaN("hello"); // true

## ✨ 9. Function Methods

js

CopyEdit

function greet(name) {

return `Hello, ${name}`;

}

greet.length; // 1 (number of parameters)

greet.name; // "greet"

greet.toString(); // function code as string

## 🚀 10. Useful ES6+ Array & Object Shortcuts

### Destructuring

js

CopyEdit

let [a, b] = [1, 2]; // a = 1, b = 2

let {name, age} = obj; // extract from object

### Spread / Rest

js

CopyEdit

let arr2 = [...arr, 6]; // spread

function sum(...args) { // rest

return args.reduce((a,b) => a + b);

}

## 🧾 Summary Table (by Type)

| Type | Common Methods |
| --- | --- |
| Number | toFixed, toString, parseInt, Math.\* |
| String | slice, split, trim, toUpperCase, replace, includes |
| Array | push, pop, map, filter, reduce, forEach, slice, splice |
| Object | keys, values, entries, assign, hasOwn |
| Date | getFullYear, getMonth, getDate, toISOString |
| JSON | JSON.stringify, JSON.parse |
| Utility | typeof, Array.isArray, isNaN, parseInt, parseFloat |

🔁 Code with let:

js

CopyEdit

for (let i = 0; i < 3; i++) {

setTimeout(() => console.log(i), 1000);

}

#### 🧠 What happens:

* let is **block-scoped**.
* Every iteration creates a **new copy** of i that's local to that block.
* So when setTimeout executes after 1 second, each arrow function “remembers” the correct i it had during that loop iteration.

#### ✅ Output (after 1 second):

CopyEdit

0

1

2

### 🔁 Code with var:

js

CopyEdit

for (var i = 0; i < 3; i++) {

setTimeout(() => console.log(i), 1000);

}

#### 🧠 What happens:

* var is **function-scoped**, not block-scoped.
* There is only **one shared** i across all iterations.
* When the setTimeout callbacks finally run, the loop has already finished, and i === 3.

#### ❌ Output (after 1 second):

CopyEdit

3

3

3

### 🧪 How to fix the var version (if you must use var)

Use an IIFE (Immediately Invoked Function Expression):

js

CopyEdit

for (var i = 0; i < 3; i++) {

(function(j) {

setTimeout(() => console.log(j), 1000);

})(i);

}

This captures the value of i in j during each iteration.

#### ✅ Output:

CopyEdit

0

1

2

### 🔚 Summary:

| Declaration | Output | Reason |
| --- | --- | --- |
| let | 0, 1, 2 | Block-scoped, new i each time |
| var | 3, 3, 3 | Function-scoped, one shared i |

### ✅ 1. ****Reconciliation in React****

**Reconciliation** is the process React uses to **update the DOM efficiently**.

**In simple terms**:  
When your component’s state or props change, React doesn’t immediately change the actual browser DOM. Instead:

* It **creates a virtual DOM** (a lightweight copy of the real DOM).
* Then it **compares** (diffs) the new virtual DOM with the previous one.
* It calculates the **minimal set of changes** needed.
* Finally, it **updates the real DOM** in the most efficient way possible.

#### Why is it important?

Reconciliation helps improve performance by avoiding unnecessary DOM operations.

#### Real example:

jsx

CopyEdit

function App() {

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

return <button onClick={() => setCount(count + 1)}>Clicked {count} times</button>;

}

Here, when count changes, only the **text inside the button** changes. React skips re-rendering the entire DOM—**thanks to reconciliation**.

### ✅ 2. ****React Event Loop****

React itself doesn’t **create** a new event loop; it **uses JavaScript’s Event Loop** (since React runs in the browser).

#### 🔄 JavaScript Event Loop Quick Recap:

* JS is **single-threaded**, meaning it executes one task at a time.
* The **Event Loop** coordinates:
  + **Call stack**: current function execution.
  + **Web APIs**: like setTimeout, DOM events.
  + **Callback queue (macrotasks)**: tasks like setTimeout.
  + **Microtask queue**: tasks like Promise.then.

#### React + Event Loop:

React hooks like useEffect, state updates like setState, and DOM events (onClick, etc.) are handled using this event loop.

js

CopyEdit

console.log("start");

setTimeout(() => console.log("timeout"), 0);

Promise.resolve().then(() => console.log("promise"));

console.log("end");

// Output:

// start

// end

// promise

// timeout

React **batch updates state** inside event loop phases to reduce re-renders and optimize performance.

### ✅ 3. ****Frontend vs Backend (in context of Event Loop)****

#### Frontend (React / Browser):

* Uses **JavaScript event loop** in the browser.
* Handles **DOM events**, **UI updates**, **XHR/fetch requests**, etc.
* Has access to **Web APIs** like setTimeout, fetch, localStorage.

#### Backend (Node.js):

* Also uses an **event loop** (same JavaScript model), but managed by Node.
* Handles **HTTP requests**, **file system**, **database operations** using **non-blocking I/O**.
* Node has **its own event loop system** with a bit different structure:
  + Uses **libuv** under the hood.
  + Handles tasks like fs.readFile(), DB calls, etc., off the main thread.

### 📌 Summary

| Concept | React / Frontend | Backend (Node.js) |
| --- | --- | --- |
| **Reconciliation** | Virtual DOM diffing and updating | ❌ Not applicable |
| **Event Loop** | JS browser event loop | Node.js event loop via libuv |
| **Use Case** | UI interactions, rendering | Handling requests, DB, file I/O |