JAVASCRIPT

Q1. What is JavaScript?
Answer: JavaScript is a high-level, interpreted programming language used to make web pages interactive.
Example: alert('This is JS Popup'); //Open a pop up alert msg.

Q2. Difference between var, let, and const?
Answer:
Var - var is function-scoped, can be redeclared and updated, and is hoisted with an initial value of undefined.
Let - let is block-scoped, can be updated but not redeclared in the same scope, and is hoisted but not initialized until execution.
Const - const is also block-scoped, must be initialized at the time of declaration, and cannot be updated or redeclared, though the contents of objects or arrays declared with const can still be modified.

Q3. What are arrow functions?
Answer: Arrow functions are a concise way to write functions in JavaScript using the => syntax, introduced in ECMAScript 6 (ES6).
Example:
```
// Traditional function
function add(a, b) {
    return a + b;
}
// Arrow function (concise)
const add = (a, b) => a + b;
```

Q4. What is hoisting?
Example: Hoisting in JavaScript is the behavior where variable and function declarations are moved to the top of their scope before the code is executed.
```
console.log(a); // undefined
var a = 10;

sayHello(); // "Hello!"
function sayHello() {
   console.log("Hello!");
}
```

Q5. What is a callback function?
Answer: In React (and JavaScript in general), a callback function is simply a function that is passed as an argument to another function and is executed later, usually after some event or operation occurs.
Example:
```
function greet(name, callback) {
  console.log("Hello, " + name);
  callback(); // calling the callback function
}

function sayGoodbye() {
  console.log("Goodbye!");
}
// Passing sayGoodbye as a callback
greet("Amit", sayGoodbye);
```

Q6. What is an IIFE?
Answer: Immediately Invoked Function Expression - a function that runs as soon as it's defined.
Example:
```
(function() {
  console.log('IIFE');
})();
```

Q7. What is the use of setTimeout()?
Answer: Executes code after a delay.
Example: setTimeout(() => console.log('Delayed'), 1000);

Q8. Explain event bubbling.
Answer: Event bubbling means that when an event happens on an element, it first runs on that element and then passes upward to its parent, grandparent, and so on up to the document. It is the default event flow in JavaScript.
Example:
```
<div id="parent">
  <button id="child">Click Me</button>
</div>

<script>
  document.getElementById("parent").addEventListener("click", function () {
    console.log("Parent clicked");
  });

  document.getElementById("child").addEventListener("click", function () {
    console.log("Child clicked");
  });
</script>
```

Q9. What is the difference between null and undefined?
Answer:
Null - null is an assigned value meaning "no value".
Undefined - undefined means a variable has been declared but not assigned.

Q10. What are Promises?
Answer: In JavaScript, Promises are a special object used to handle asynchronous operations (like fetching data from a server, reading a file, etc.).
It has 3 states:
- Pending → Initial state, operation is still running.
- Fulfilled (Resolved) → Operation completed successfully, and we got the result.
- Rejected → Operation failed, and we got an error.

Example:
```
const myPromise = new Promise((resolve, reject) => {
  let success = true;
  if (success) {
      resolve("Operation successful!"); // fulfilled
  } else {
      reject("Operation failed!"); // rejected
  }
});

myPromise
  .then(result => console.log(result)) // runs if resolved
  .catch(error => console.log(error)) // runs if rejected
  .finally(() => console.log("Done!")); // runs always
```

Q11. What is async/await?
Answer: Async and Await are used to simplify handling asynchronous operations using promises.

- **Async -** The **async** keyword is used to declare a function that always returns a Promise.
- **Await -** The **await** keyword is used inside an async function to pause execution until a Promise is resolved or rejected.

This makes asynchronous code look and behave more like synchronous code, improving readability and reducing the need for chaining .then() or .catch().
Async/await also makes error handling easier using try...catch blocks.

Example:
```
async function fetchData() {
 try {
  const response = await fetch("https://api.example.com/data");
  const data = await response.json();
  console.log(data);
 } catch (error) {
  console.error("Error fetching data:", error);
 }
}
```

Q12. What are higher-order functions?
Answer: A higher-order function (HOF) is a function that either:
1. Take another function as an argument, or
2. Returns a function as its result.

Example:
Passing a function as argument:
```
function greet(name) {
 return "Hello, " + name;
}

function processUserInput(callback) {
 let name = "Himanshu";
 return callback(name);
}
console.log(processUserInput(greet));  // Output: Hello, Himanshu
```

Returning a function:
```
function multiplier(factor) {
 return function(num) {
  return num * factor;
 };
}
let double = multiplier(2);
console.log(double(5)); // 10
```

Q13. What is destructuring?
Answer: Extracting values from arrays or objects into variables.
let [a, b] = [1, 2];
let {name} = {name: 'Rachit', age: 25};

Q14. Difference between map() and forEach() in JavaScript.
Answer:
map() - Returns a new array of transformed elements. Used when you need to modify data and use the result later.
[1,2,3].map(x => x * 2); // [2,4,6]

forEach() - Does not return anything (returns undefined).
[1,2,3].forEach(x => x * 2); // undefined

Q15. What is the spread operator?
Answer: The JavaScript spread operator (...) is allow us to quickly copy all **or** part of an existing array or object into another array or object **(in ES6)**.
Example:
const arr = [1, 2];
const newArr = [...arr, 3]; // [1, 2, 3]

Q16. What is event delegation in JavaScript?
Answer: Event delegation is a technique in JS where instead of adding event listener to multiple child elements, we add a single event listener to their parent element. This works because of event bubbling – when an event occurs on a child element, it "bubbles up" through its parent elements in the Dom tree.
Example:
```
<ul id='itemList'>
  <li>Apple</li>
  <li>Banana</li>
  <li>Cherry</li>
</ul>

//JavaScript
const list = document.getElementById('itemList');
list.addEventListener('click', function(e) {
   if (e.target.tagName === 'LI') {
      console.log('Item clicked:', e.target.textContent);
   }
});
```

Q17. What is the difference between function declaration and function expression?
Answer: Function Declaration: A function declaration defines a named function using the function keyword.
Example:
```
sayHi(); // Works (because of hoisting)
function sayHi() {
  console.log("Hi there!");
}
```

Key Points:
- **Hoisted → You can call the function before it is defined in the code.**
- **The function name is mandatory.**
- **Stored in memory during the compile phase.**

Function Expression: A function expression defines a function and assigns it to a variable (or constant).
```
const greet = function() {
  console.log("Hello!");
};
```

Key points:
- **Not hoisted → You cannot call it before it's defined.**
- **Can be anonymous (no name) or named.**
- **Created during the execution phase (when that line runs).**

Q18. What is a pure function?
Answer: A function that produces no side effects and returns the same output for the same input.
Example:
```
function add(a, b) {
   return a + b;
}
```

Q19. Explain the difference between slice and splice.
Answer:
slice() returns a shallow copy without modifying the original array.
splice() modifies the array in place.

```
let arr = [1, 2, 3, 4];
console.log(arr.slice(1, 3));   // [2, 3]
arr.splice(1, 2);               // arr is [1, 4]
```

Q20. What is the event loop in JavaScript?
Answer: The mechanism that handles execution of multiple chunks of code like callbacks, events, etc.

```
console.log('Start');
setTimeout(() => console.log('Timeout'), 0);
console.log('End');
```

Q21. What is a promise chain?
Answer: promise chain in jS is a way to execute multiple asynchronous operations in sequence, where each operation starts only after the previous one has completed successfully. It's done by returning a new promise inside a .then() block and continuing the chain with another .then()

```
fetch('/api')
  .then(res => res.json())
  .then(data => console.log(data));
  .catch(error => console.log(error));
```

Q22. What is optional chaining (?.) in JS?
Answer: Allows safe access to deeply nested properties.

```
let user = {};
console.log(user?.profile?.email); // undefined, no error
```

Q23. What is nullish coalescing (??)?
Answer: Returns the right-hand side if the left is null or undefined.

```
let name = null;
console.log(name ?? 'Guest'); // 'Guest'
```

Q24. Difference between synchronous and asynchronous?

- Synchronous: tasks run sequentially.
- Asynchronous: tasks can run in the background.

```
console.log('1');
setTimeout(() => console.log('2'), 0);
console.log('3');
```

Explanation: Output: 1, 3, 2 — setTimeout is async.

Q25. What is the difference between for...of and for...in?

- for...in iterates over object keys.
- for...of iterates over iterable values (arrays, strings).

```
for (let key in {a:1, b:2}) console.log(key); // a, b
for (let val of [1,2]) console.log(val); // 1, 2
```

Q26. What is memoization?
Answer: Memoization is an optimization technique used to speed up functions by caching the results of expensive function calls.
If the function is called again with the same inputs, it returns the cached result instead of recalculating.

Example:

```
function memoizedSquare() {
  const cache = {}; // stores previous results
  return function(num) {
    if (cache[num]) {
      console.log("Getting from cache...");
      return cache[num];
    }
    console.log("Calculating...");
    const result = num * num;
    cache[num] = result;
    return result;
  };
}

const square = memoizedSquare();
console.log(square(4)); // Calculating... → 16
console.log(square(4)); // Getting from cache... → 16
console.log(square(5)); // Calculating... → 25
```

Q27. What is debouncing in JavaScript/React?
Answer: Debouncing is a performance optimization technique that ensures the function will only run after a certain time has passed since the last time it was invoked. It prevents a function from running too often.

Example:

```
import { useState, useEffect } from "react";

function Search() {
  const [query, setQuery] = useState("");
  const [debouncedQuery, setDebouncedQuery] = useState(query);

  useEffect(() => {
```

```
  const handler = setTimeout(() => {
    setDebouncedQuery(query);
  }, 500); // delay of 500ms

  return () => {
    clearTimeout(handler); // cleanup on each keystroke
  };
 }, [query]);

 useEffect(() => {
  if (debouncedQuery) {
    console.log("API call with:", debouncedQuery);
  }
 }, [debouncedQuery]);

 return <input onChange={(e) => setQuery(e.target.value)} placeholder="Search..." />;
}
```

Q28. What is throttling?
Answer: Throttling is a performance optimization technique that ensures a function is called at most once in a specified time interval - even if it's triggered multiple times.

```
import { useState } from "react";

function ThrottleButton() {
 const [count, setCount] = useState(0);
 let lastClick = 0;

 const handleClick = () => {
  const now = Date.now();
  if (now - lastClick > 1000) { // allow click once per second
    setCount(count + 1);
    lastClick = now;
  }
 };

 return (
  <div>
    <button onClick={handleClick}>Click me</button>
    <p>Count: {count}</p>
  </div>
 );
}

export default ThrottleButton;
```

Q29. What are data attributes in HTML/JS?
Answer: In HTML and JavaScript, data attributes are custom attributes you can add to HTML elements to store extra information that doesn't have a visual representation on the page. They are very handy for passing data from HTML to JavaScript without needing additional DOM elements or global variables.

Example: John Doe
data-id="123" → stores a user ID
data-role="admin" → stores the role

Accessing data attributes in JavaScript

1. **Using dataset property**
   ```
   const userDiv = document.getElementById("user");
   // Access data attributes
   console.log(userDiv.dataset.id); // "123"
   console.log(userDiv.dataset.role); // "admin"
   // Modify data attributes
   userDiv.dataset.role = "superadmin";
   console.log(userDiv.dataset.role); // "superadmin"
   ```

   dataset automatically converts kebab-case to camelCase.
   Example: data-user-name → dataset.userName.

2. **Using getAttribute / setAttribute**
   ```
   console.log(userDiv.getAttribute("data-id")); // "123"
   userDiv.setAttribute("data-role", "editor");
   console.log(userDiv.getAttribute("data-role")); // "editor"
   ```

Q30. What are the different types of errors in JS?
Answer: SyntaxError, ReferenceError, TypeError // ReferenceError console.log(x); // x is not defined

Q31. How to handle errors OR error boundaries in JS?
Answer: Error boundaries help you to catch Javascript error anywhere in the child components. They are most used to log the error and show a fallback UI.
Using try...catch block.

```
try {
  throw new Error("Something went wrong");
} catch (e) {
  console.log(e.message);
}
```

Q32. What is this in JavaScript?
Answer: Refers to the current execution context.

```
const obj = {
  name: 'Rachit',
  getName() {
    return this.name;
  }
};
```

Q33. How does bind() work?
Answer: Returns a new function with a bound this.
Example:

```
let greet = function() {
  console.log(this.name);
};
let person = {
  name: 'Rachit'
};
let bound = greet.bind(person);
bound(); // Rachit
```

Q34. What is the difference between call() and apply()?
Answer: In JavaScript, call() and apply() are function methods used to invoke a function immediately while explicitly setting the this context. The only difference is how arguments are passed.

Example:

```
function greet(city, country) {
 console.log(`Hello, I'm ${this.name} from ${city}, ${country}`);
}
const person = { name: "Himanshu" };

// call() — pass arguments individually
greet.call(person, "Indore", "India");

// apply() — pass arguments as an array
greet.apply(person, ["Indore", "India"]);
```

Q35. What is a generator function?
Answer: A generator function is a special function defined using **function*** that can pause and resume its execution. It uses the **yield** keyword to return values one by one and automatically creates an iterator. Generator are useful for handling large data sequences and implementing custom iterators.
It is useful for async flow(Before async/await)

Example:

```
function* number() {
   yield 1;
   yield 2;
}

const gen = numbers();
console.log(gen.next()); // { value: 1, done: false }
console.log(gen.next()); // { value: 2, done: false }
console.log(gen.next()); // { value: undefined, done: true }
```

Example2:

```
function* idGenerator() {
 let id = 1;
 while (true) {
   yield id++;
 }
}

const gen = idGenerator();

console.log(gen.next().value); // 1
console.log(gen.next().value); // 2
console.log(gen.next().value); // 3
```

Q36. What is Construction function?
Answer: Used to create instances from constructor functions.
```
function Person(name) {
  this.name = name;
}
let p = new Person('Rachit');
```

Q37. What is strict mode in JS?
Answer: A way to catch common coding mistakes by enabling "use strict".
```
"use strict";
 x = 10; // ReferenceError: x is not defined
```

Q38. What are rest parameters?
Answer: Allows a function to accept an indefinite number of arguments as an array.
```
function sum(...nums) {
  return nums.reduce((a, b) => a + b);
}
```

Q39. How to clone an object?
Answer: Using Object.assign() or spread operator.
```
let clone = { ...original };
```

Q40. What is the difference between deep and shallow copy?
Answer: Shallow copy copies references of nested objects.
Deep copy copies everything recursively.
Example:
```
let obj = { a: { b: 1 } };
let shallow = { ...obj };
obj.a.b = 2;
console.log(shallow.a.b); // 2
```

Q41. What is the typeof null?
Answer: It returns 'object', which is a known bug in JavaScript.
```
console.log(typeof null); // 'object'
```

Q42. What are falsy values in JS?
Answer: Values that evaluate to false in Boolean context.
Falsy Values: ● false, 0, '', null, undefined, NaN

Q43. How to check if a variable is an array?
Answer: Use Array.isArray().
```
Array.isArray([1, 2, 3]); // true
```

Q44. What are JavaScript modules?
Answer: Files that export/import functionality using export and import.
```
// utils.js
export const add = (a, b) => a + b;

// app.js
import { add } from './utils.js';
```
//Explanation; Encapsulates code and promotes reusability.

Q45. What is function currying?
Answer: Transforming a function that takes multiple arguments into nested functions.
```
function curry(a) {
  return function(b) {
    return a + b;
  };
}
curry(2)(3); // 5
```

Q46. What are default parameters?
Answer: Set default values for function parameters.
```
function greet(name = 'Guest') {
  return `Hello ${name}`;
}
```

Q47. What is the difference between == and Object.is()?
Answer: Object.is() is more accurate for edge cases like NaN.
```
NaN == NaN // false
Object.is(NaN, NaN) // true
```
Explanation: Object.is() compares more precisely.

Q48. What is call stack in JavaScript?
Answer: A stack data structure used to keep track of function calls.

Q49. How do you compare objects in JS?
Answer: Use deep comparison (e.g., JSON method or libraries).
JSON.stringify(obj1) === JSON.stringify(obj2);
Explanation: Works for flat objects, not good for nested objects with different key order.

Q50. What is destructuring assignment?
Answer: Extract values from arrays or objects into variables.
```
let {name} = {
  name: 'Rachit'
};
let [a, b] = [1, 2];
```

Q51. What is optional parameter in functions?
Answer: Parameters not required in function calls.
```
function greet(name) {
  return `Hello ${name || 'Guest'}`;
}
```

Q52. How to convert a string to a number?
Answer: Use Number(), unary +, or parseInt().
```
+'42'; // 42
Number('42'); // 42
parseInt('42'); // 42
```

Q53. What is the purpose of Array.prototype.reduce()?
Answer: Used to reduce an array to a single value.
[1,2,3].reduce((a, b) => a + b, 0); **// 6**

Q54. What is dynamic typing in JS?
Answer: Variables can hold values of any type and change types at runtime.

Q55. What is the new keyword in JS?
Answer: The new keyword is used to create an instance of an object from a constructor function or class.
```
function Person(name) {
  this.name = name;
}
const p1 = new Person("Himanshu");
```

Q56. What are template engines in JS?
Answer: Template engines allow us to generate dynamic HTML using JavaScript. They replace placeholders in templates with real values.
Examples: EJS, Handlebars, Pug

Q57. How does JSON.parse() and JSON.stringify() work?
Answer:
JSON.stringify() - Converts JS object → JSON string
JSON.parse() - Converts JSON string → JS object

Q58. What is a tagged template literal?
Answer: A tagged template allows calling a function with template literal values for custom formatting.
Example:
```
function tag(strings, value){
  return strings[0] + value.toUpperCase();
}
tag`hello ${"world"}`; // hello WORLD
```

Q59. What is Promise.all()?
Answer: Runs multiple promises in parallel and returns when all succeed or returns an error if any fails.
Example: Promise.all([p1, p2, p3]);

Q60. What are global variables?
Answer: Variables declared outside any function, accessible everywhere.

Q61. How to prevent modification of an object?
Answer: Use Object.freeze().

Q62. What is a WeakMap?
Answer: WeakMap stores key-value pairs where keys must be objects. It allows automatic garbage collection when the object key has no other references, making it useful for private data and memory-efficient storage.
Example:
```
const weakMap = new WeakMap();
let obj = { name: "John" };
weakMap.set(obj, "Some private value");
obj = null;  // object removed // weakMap no longer keeps it in memory automatically
```
Q63. What is a Symbol in JavaScript?
Answer: A Symbol is a primitive data type in JavaScript introduced in ES6, used to create unique and immutable identifiers. Even if two symbols have the same description, they are always different values, which makes them useful for creating private or unique object keys without risk of conflicts.

Example:

```
const id1 = Symbol("id");
const id2 = Symbol("id");

console.log(id1 === id2); // false (unique values)

const user = {};
user[id1] = "Himanshu";  // unique private key
```

Q64. What is prototype chaining?

Answer: Prototype chaining allows objects to inherit features from other objects. If a property is not found on the object itself, JavaScript looks for it on its prototype chain.

Example:

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

// Here, child does not have its own greet method, but it finds it in its prototype (parent) using prototype
```

65. What is event loop and microtask queue?

Answer:

Event Loop: Event Loop handles asynchronous callbacks and manages execution between the call stack and callback queues.

Microtask Queue: The Microtask Queue stores callbacks from Promises (then, catch, finally) and MutationObservers. When JavaScript finishes executing the current synchronous code in the call stack, it checks the Microtask Queue first and runs all microtasks before moving to the Macrotask Queue and before rendering the UI. This is why promise callbacks run faster and with higher priority than setTimeout or other macrotasks.

Example:

```
console.log("Start");
Promise.resolve().then(() => console.log("Microtask"));
setTimeout(() => console.log("Macrotask"), 0);
console.log("End");
```

Output:

```
Start
End
Microtask   // runs first (microtask queue)
Macrotask   // runs later (macrotask queue)
```

Q66. What is the difference between Object.seal() and Object.freeze()?

Answer:

Object.seal() – Allows modifying existing property values but **prevents adding or deleting** properties from an object.

Object.freeze() - Prevents modification of existing properties and prevents adding/removing properties.

Example:

```
const obj = Object.freeze({ name: 'Rachit' });
obj.name = 'New'; // Won't change
```

Q67. What is a polyfill?

Answer: A polyfill is a piece of code (usually JavaScript) that adds support for features that are not available in older browsers. It provides a fallback implementation so that new features (like Promise, Array.from, etc.) work even where they are not natively supported.

Q68. What is a thunk?

Answer: A thunk is a function that wraps another function and delays its execution. In JavaScript, thunks are often used for lazy evaluation or in Redux for handling async logic.

```
const thunk = () => {
  return fetchData();  // executed later
};
// usage
thunk();
```

Q69. What is the instanceof operator?

Answer: The instanceof operator is used to check whether an object is an instance of a specific constructor function or class. It returns true if the object exists in the prototype chain of that constructor, otherwise false.

Example:

```
function Animal() {}
const dog = new Animal();

console.log(dog instanceof Animal); // true
console.log(dog instanceof Object); // true
console.log(dog instanceof Array);  // false
```

Q70. What is a factory function?

Answer: A factory function is a function that returns a new object. It is used to create multiple similar objects without using classes or constructors, and it supports closures for private data.

Example:

```
function createUser(name, age) {
  return {
    greet() {
      console.log(`Hello, I am ${name}`);
    }
  };
}

const user1 = createUser("Alex", 25);
const user2 = createUser("John", 30);
user1.greet(); // Hello, I am Alex
user2.greet(); // Hello, I am John
```

Q71. What is a callback hell?

Answer: Callback hell happens when asynchronous operations are nested inside each other, making code hard to read and maintain. It usually appears as deeply indented “pyramid” code. This can be avoided using Promises or async/await.

Example:

```
getUser(id, function(user) {
   getOrders(user.id, function(orders) {
      getOrderDetails(orders[0].id, function(details) {
         console.log(details);
      });
   });
});

getUser(id)
  .then(user => getOrders(user.id))
  .then(orders => getOrderDetails(orders[0].id))
  .then(details => console.log(details))
  .catch(err => console.error(err));
```

Q72. What is lexical scope?

Answer: Lexical scope in JavaScript means that a function’s scope is determined by where it is defined in the source code, not where it is called. Variables defined in a parent scope are accessible to its child functions, but not vice versa. This concept allows closures to access variables from their outer functions.

Example:

```
function outer() {
   let name = "Himanshu";
   function inner() {
      console.log(name); // can access 'name' from outer scope
   }
   inner();
}
outer();
```

Q73. What is npm init and npm init -y.

Answer: npm init and npm init -y are commands used in Node.js projects to create a package.json file, which stores metadata about your project and its dependencies.

npm init: Starts an interactive setup. Asks you a series of questions, such as: Project name, Version, Description, Entry point (index.js), Author, License
Based on your answers, it generates a package.json.
npm init -y: Creates a package.json automatically. Skips all questions and uses default values. Much faster for quick setups or experiments.

Q74. Create express server in NodeJS.

Answer: For creating project, first choose file directory and run the below commands:

```
npm init -y
npm install express cors
```

after that write this code in index.js file, now server is up and running on 5000 port.

index.js file

```
const express = require('express')
const cors = require('cors')
const app = express();
const port = 5000;

app.use(cors());
app.use(express.json());
const serverFunction = (req, res) => {
  res.send({
    success: true,    msg: "Server is started"
  })
}
app.get("/", serverFunction);

app.listen(port, () => {
  console.log('server is running')
})
```

React

Q1. What is React?
Answer: React is a JavaScript library for building user interfaces (UIs), especially for single-page applications (SPAs). It was developed by Facebook (now Meta). Instead of manipulating the browser's DOM directly, React creates a virtual DOM that it updates efficiently. Key features:
- Virtual DOM for efficient rendering.
- Component-based architecture.
- Unidirectional data flow.
- JSX for writing UI in JavaScript.
- Hooks for functional components.

Example:
```
function Hello() {
  return <h1>Hello, React!</h1>;
}
```

Q2. What are the main features of React?
Answer: React is a component-based library that build reusable and maintainable UI elements. It uses a Virtual DOM for efficient rendering, updating only what's necessary, which improves performance. With JSX, we can write HTML-like code in JavaScript, making the UI code more readable. React follows one-way data flow, keeping state predictable, and Hooks allow functional components to manage state and lifecycle logic easily. Its declarative nature and strong ecosystem make it ideal for building scalable web applications.

Q3. What is JSX? How is it different from HTML?
Answer: JSX stands for JavaScript XML. JSX is a JavaScript syntax extension that allows us to write HTML-like structures within JavaScript code, making the code more readable and declarative, whereas HTML is a markup language designed for structuring web pages.

How is JSX different from HTML?
1. Syntax & Integration: JSX blends HTML-like syntax with JavaScript, enabling dynamic content and logic within markup, whereas HTML is a static markup language.
2. Compilation: Browsers cannot directly interpret JSX; it must be converted into JavaScript, whereas HTML is natively understood by browsers.
3. Attributes & Naming: JSX uses camelCase for attributes (e.g., className instead of class), while HTML uses lowercase attributes.
4. Expression Support: JSX allows embedding JavaScript expressions using {} within elements, whereas HTML does not support direct JavaScript execution inside markup.

Q4. What is the Virtual DOM, and how does React use it?
Answer: Virtual DOM is a lightweight copy of the real DOM (Document Object Model) used by React to improve performance and efficiency when updating the user interface. It's a virtual copy kept in memory that React uses to track changes and update the actual DOM efficiently.

Q5. Explain the difference between functional and class components.
Answer: In React, class components are ES6 classes that extend React.Component and can have their own state and lifecycle methods, like componentDidMount or componentDidUpdate. They were traditionally used when you needed to manage state or handle side effects.
Functional components, on the other hand, are simple JavaScript functions that return JSX. Initially, they were "stateless," but with the introduction of Hooks (useState, useEffect, etc.), functional components can now handle state and lifecycle logic just like class components, often in a cleaner and more concise way.
In practice, functional components are preferred today because they are simpler, easier to read, easier to test, and lead to less boilerplate code. Class components are still valid but are mostly used in older codebases.

Example:

```
function Welcome(props) {
  return <h1>Hello, {props.name}</h1>;
}


class Welcome extends React.Component {
  render() {
    return <h1>Hello, {this.props.name}</h1>;
  }
}
```

Q6. Explain the use of `key` in React lists.
Answer: In React lists, the key attribute is a special string used to uniquely identify each element within a list, helping React efficiently update and re-render the list when items are added, removed, or changed.

Example:
```
import { useState } from "react";

function Lists() {
  const items = ["Apple", "Banana", "Cherry"];
  return(
    <ul>
      {
  items.map((item, index) =>
<li key={index}>{item}</li>
  )
      }
    </ul>
  );
}
```

Q7. React Routers and how we use them?
Answer: React Router is a standard library for routing in React applications. It allows us to navigate between different components/pages without reloading the entire page, enabling a single-page application (SPA) behavior.

command: npm install react-router-dom

```
import { BrowserRouter as Router, Routes, Route, Link } from 'react-router-dom';
import Home from './Home';
import About from './About';

function App() {
 return (
  <Router>
   <nav>
    <Link to="/">Home</Link>
    <Link to="/about">About</Link>
   </nav>

   <Routes>
    <Route path="/" element={<Home />} />
    <Route path="/about" element={<About />} />
   </Routes>
  </Router>
 );
}
```

Q8. Difference between useEffect and useLayoutEffect.
Answer:
Both useEffect and useLayoutEffect are React hooks used to perform side effects after a component renders.
useEffect runs after the render has been committed to the screen, making it ideal for tasks like data fetching, subscriptions, or logging.
useLayoutEffect runs synchronously after the DOM updates but before the browser repaints, ensuring the UI is updated before the user sees any visual changes. This makes useLayoutEffect suitable for operations like measuring DOM elements or adjusting layout.
In summary, use useEffect for non-blocking side effects and useLayoutEffect for layout-related DOM manipulations.

Q9. What are props in React? How are they different from state?
Answer: In React, props (short for properties) are read-only values passed from a parent component to a child component. They allow you to make components reusable and dynamic by letting the parent control data and behavior of the child. For example, you might pass a title or onClick function as a prop.

State is internal to a component and represents data that can change over time. A component can update its own state using setState (in class components) or useState (in functional components), and when the state changes, React automatically re-renders the component.

Key difference:
1. Props are immutable — the child cannot change them.
2. State is mutable — the component owns it and can update it.

In short: props are for passing data in the child component, state is for managing data inside a component.

Example of Props:

```
function Greeting(props) {
 return <h1>Hello, {props.name}!</h1>;
}

function App() {
 return <Greeting name="Alice" />;  // Passing "Alice" as a prop
}
```

Example of State:

```
import { useState } from "react";

function Counter() {
 const [count, setCount] = useState(0);  // State variable

 return (
  <div>
   <p>Count: {count}</p>
   <button onClick={() => setCount(count + 1)}>Increment</button>
  </div>
 );
}
```

Q10. How do you optimize react application.
Answer: To optimize a React application, we use techniques like React.memo, useMemo, and useCallback to prevent unnecessary re-renders of components and functions. We implement code splitting to break the app into smaller chunks so that only the required parts load initially, and use lazy loading for components to reduce the initial bundle size. We also avoid anonymous functions in JSX by defining handlers with useCallback to prevent creating new functions on every render. These practices together improve performance, reduce unnecessary renders, and make the app more scalable.

Q11. What is pure function in react.
Answer: In React, a pure function usually refers to a function (often a component) that always produces the same output for the same input and has no side effects.

Q12. What is the difference between controlled and uncontrolled components?
Answer: In React, a controlled component is a form element (like <input> or <textarea>) whose value is controlled by React state. The component's value is set via state and updated using onChange handlers. This gives you full control over the form data and makes it easy to validate, manipulate, or submit.

```
import { useState } from "react";

function ControlledForm() {
 const [name, setName] = useState("");

 const handleSubmit = (e) => {
  e.preventDefault();
  alert(`Controlled Input: ${name}`);
 };

 return (
  <form onSubmit={handleSubmit}>
   <label>
    Name:
    <input
     type="text"
     value={name} // value comes from React state
     onChange={(e) => setName(e.target.value)} // update state on change
    />
   </label>
   <button type="submit">Submit</button>
  </form>
 );
}

export default ControlledForm;
```

Uncontrolled component: An uncontrolled component is a form element that manages its own state internally, just like in plain HTML. React doesn't control its value; instead, you can use a ref to get the current value when needed.

```
import { useRef } from "react";

function UncontrolledForm() {
 const nameRef = useRef();

 const handleSubmit = (e) => {
  e.preventDefault();
  alert(`Uncontrolled Input: ${nameRef.current.value}`);
 };

 return (
  <form onSubmit={handleSubmit}>
   <label>
    Name:
    <input
     type="text"
     ref={nameRef} // DOM manages the input
    />
   </label>
   <button type="submit">Submit</button>
  </form>
 );
}

export default UncontrolledForm;
```

Q13. What is CSRF attack and how do you prevent it?
Answer: CSRF (Cross-Site Request Forgery) is an attack where an attacker tricks your browser into sending unauthorized requests to a trusted site while you're logged in. Since the browser automatically includes your cookies, the attacker's request runs with your session. It can be prevented using SameSite cookies, CSRF tokens, and Origin/Referer header checks.
Q14. What is CORS?
Answer: CORS (Cross-Origin Resource Sharing) is a browser security feature that controls whether a web page from one domain can access resources from another domain. Servers allow cross-origin requests by sending appropriate CORS headers like Access-Control-Allow-Origin.

Q15. What's the cleanup function in useEffect?
Answer: In React, the cleanup function in useEffect is a function you return from the effect that runs before the component unmounts or before the effect runs again on re-renders. It's used to clean up side effects like subscriptions, timers, or event listeners to prevent memory leaks.
Example:

```
useEffect(() => {
 const timer = setInterval(() => {
   console.log('tick')
   }, 1000);
 return () => clearInterval(timer);
}, []);
```

Q16. Explain about hooks in react.
Answer: Hooks are special functions in React taht allows us to use 'state, lifecycle method and other React features' from functional components. It is introduced in react 16.8, they make code cleaner and easier to maintain.
Some important hooks are:
1. useState
2. useEffect
3. useContext
4. useReducer
5. useRef
6. useMemo
7. useCallback
8. useLayoutEffect

1. useState:
The useState hook is a React Hook that allows us to manage state to functional components. It returns a state variable and a function to update it.
Example: const [state, setState] = useState(initialValue);

```
import React, { useState } from 'react';

function Counter() {
  const [count, setCount] = useState(0); // initialize count with 0

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>Increase</button>
    </div>
  );
}
```

2. useEffect:
The useEffect hook in React is used to handle side effects in functional components, such as:
a. Fetching data from an API
b. Updating the DOM manually
c. Setting up subscriptions or timers
In class components, these were handled in lifecycle methods like componentDidMount, componentDidUpdate, and componentWillUnmount.

Example:

```
import React, { useState, useEffect } from "react";

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

  // useEffect runs after every render by default
  useEffect(() => {
    console.log("Component rendered or count changed:", count);

    // Optional cleanup function
    return () => {
      console.log("Cleanup before next effect or unmount");
    };
  }, [count]); // Dependency array: effect runs only when `count` changes

  return (
    <div>
      <h1>Count: {count}</h1>
      <button onClick={() => setCount(count + 1)}>Increment</button>
    </div>
  );
}

export default Counter;
```

3. useContext:
useContext is one of the React Hooks that helps us to manage global state or share data between components without prop drilling (i.e., without passing props manually through every level of the component tree).
Example:

```
// App.js
import React, { createContext, useContext } from 'react';

// Step 1: Create a Context
const ThemeContext = createContext('light');

function App() {
  const currentTheme = "dark"; // variable instead of hardcoded value
  return (
    // Step 2: Provide a value
    <ThemeContext.Provider value={currentTheme}>
      <Toolbar />
    </ThemeContext.Provider>
```

```
  );
}

// toolbar.js
function Toolbar() {
  return <ThemedButton />;
}

// themeButton.js
function ThemedButton() {
  // Step 3: Consume the value
  const theme = useContext(ThemeContext);
  return <button>Current theme: {theme}</button>;
}
```

4. useReducer:
useReducer is a React hook used for managing state in a more structured way, especially when the state logic is complex or depends on previous state values. It works by defining a reducer function that takes the current state and an action, and returns a new state, while state updates are triggered by dispatching actions.

Example:
```
import React, { useReducer } from 'react';

const initialState = { count: 0 };

function reducer(state, action) {
  switch (action.type) {
    case 'increment':
      return { count: state.count + 1 };
    case 'decrement':
      return { count: state.count - 1 };
    case 'reset':
      return initialState;
    default:
      throw new Error('Unknown action');
  }
}

function Counter() {
  const [state, dispatch] = useReducer(reducer, initialState);

  return (
    <div>
      <p>Count: {state.count}</p>
      <button onClick={() => dispatch({ type: 'increment' })}>+</button>
      <button onClick={() => dispatch({ type: 'decrement' })}>-</button>
      <button onClick={() => dispatch({ type: 'reset' })}>Reset</button>
    </div>
  );
}
```

Q. We can achieve this reducer() using if-else condition, then why we use useReducer hook?
Answer: For simple state like a counter, if-else with useState works fine. useReducer is useful when state is complex or has multiple interdependent values, because it centralizes all state logic in one function, makes updates predictable, easier to maintain, and scales better as component grows.

5. useRef:
useRef is a React Hook that persists values across renders without causing a re-render when the value changes. It is commonly used to access DOM elements directly or to store mutable values such as timers, previous props, or other state that does not need to trigger UI updates.

Example:
```
import React, { useRef, useEffect } from 'react';

function InputFocus() {
  const inputRef = useRef(null);

  useEffect(() => {
    inputRef.current.focus(); // Automatically focuses input on mount
  }, []);

  return <input ref={inputRef} placeholder="Type here..." />;
}

export default InputFocus;
```

Q. What the meaning of 'access DOM elements directly'.
Answer: When we say “access DOM elements directly” in React using useRef, we can get a reference to an actual HTML element rendered in the browser, without relying on React state or props.

In HTML:

```
<input id="myInput" />
<script>
  const input = document.getElementById("myInput");

  input.focus(); // directly focus the input
```

In React: We don't use document.getElementById. Instead, we use useRef:

```
import { useRef } from "react";

function FocusInput() {
  const inputRef = useRef(null); // create a ref
  const handleFocus = () => {
    inputRef.current.focus(); // access the actual DOM node
  };

  return (
    <>
      <input ref={inputRef} type="text" /> {/* attach ref to input */}
      <button onClick={handleFocus}>Focus Input</button>
    </>
  );
}
```

Q. Comparing useRef vs useState.

Answer: useState is used to store data that affects the UI. When a useState value changes, the component re-renders.
useRef is used to store mutable values or access DOM elements directly. Changing a useRef value does not cause a re-render.
In short, use useState when UI needs to update, and use useRef when you just need to store or reference a value without re-rendering.

6. useMemo:

useMemo is a React hook that remembers the result of a function and only recomputes it when its dependencies change, which helps optimize performance by avoiding unnecessary calculations on every render.
Ex:

```
import React, { useState, useMemo } from 'react';

function App() {
  const [number, setNumber] = useState(0);

  const squared = useMemo(() => number * number, [number]);

  return (
    <div>
      <h1>Square: {squared}</h1>
      <input value={number} onChange={e => setNumber(+e.target.value)} />
    </div>
  );
}

export default App;
```

7. useCallback:

useCallback is a React hook that memoizes a function, i.e., it returns a cached version of the function that only changes if one of its dependencies changes. This helps prevent unnecessary re-creations of functions on every render.

```
import React, { useState, useCallback } from "react";

const Child = React.memo(({ onClick }) => {
  console.log("Child rendered");
  return <button onClick={onClick}>Click me</button>;
});

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

  // Memoized function
  const handleClick = useCallback(() => {
    setCount(c => c + 1);
  }, []);

  return (
    <div>
      <h1>Count: {count}</h1>
      <Child onClick={handleClick} />
    </div>
  );
}
export default Parent;
```

8. useLayoutEffect
useLayoutEffect is a React hook that runs immediately after the component renders but before the browser paints the screen.
It's mainly used when you need to read or modify the DOM layout before the user sees any visual changes - for example, measuring element sizes, adjusting scroll positions, or preventing layout flicker.

```
import React, { useLayoutEffect, useRef } from 'react';

function Box() {
  const boxRef = useRef();

  useLayoutEffect(() => {
    console.log(boxRef.current.getBoundingClientRect()); // measure element size before paint
  }, []);

  return <div ref={boxRef} style={{ width: '200px', height: '200px', background: 'skyblue' }} />;
}

export default Box;
```

Q17. Explain React.Memo(HOC).
Answer: React.memo is a higher-order component that memoizes a functional component. It prevents the component from re-rendering if its props have not changed since the previous render. It is used to optimize performance for functional components by avoiding unnecessary renders.

```
import React, { useState, useCallback } from "react";

// Child component wrapped with React.memo
const Child = React.memo(({ onClick }) => {
  console.log("Child rendered");
  return <button onClick={onClick}>Click me</button>;
});

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

  // Memoized function to prevent Child re-render
  const handleClick = useCallback(() => {
    setCount(c => c + 1);
  }, []);

  return (
    <div>
      <h1>Count: {count}</h1>
      <Child onClick={handleClick} />
    </div>
  );
}

export default Parent;
```

Q18. What is a custom hook?
Answer: A custom hook is a user-defined hook that starts with use and allows reuse of stateful logic.

Example:

```
// src/hooks/useCounter.js
import { useState } from "react";

function useCounter(initialValue = 0) {
  const [count, setCount] = useState(initialValue);

  const increment = () => setCount(prev => prev + 1);
  const decrement = () => setCount(prev => prev - 1);
  const reset = () => setCount(initialValue);

  return { count, increment, decrement, reset };
}

export default useCounter;

// src/components/Counter.js
import React from "react";
import useCounter from "../hooks/useCounter";
```

```
function Counter() {
  const { count, increment, decrement, reset } = useCounter(5); // start from 5
 return (
   <div>
     <h1>Count: {count}</h1>
     <button onClick={increment}>+1</button>
     <button onClick={decrement}>-1</button>
     <button onClick={reset}>Reset</button>
   </div>
 );
}

export default Counter;
```

Q19. What is redux toolkit and redux architecture?
Answer: Redux is a state management library where the entire app's data is stored in a single store. Whenever we need to change the state, we dispatch an action, which goes through a reducer to return the new state - this process follows a unidirectional data flow.
Redux Toolkit is its modern version that simplifies the setup - using tools like createSlice and configureStore, we can easily write reducers and actions without extra boilerplate code.

Q20. Authentication providers.
Answer: Authentication providers are services or systems that verify a user's identity and manage login, signup, and session handling. They can be third-party platforms like Firebase, Auth0, AWS Cognito, or custom solutions using JWT, and may support social, enterprise, or custom login methods.

Examples of Common Authentication Providers:
1. Firebase Authentication – Google's service supporting email/password, phone, and social logins.
2. Auth0 – A popular third-party platform for secure authentication and authorization.
3. AWS Cognito – Amazon's identity service for managing user authentication and tokens.
4. Okta – Enterprise-grade identity management service.
5. NextAuth.js – Authentication library for Next.js that supports multiple providers.
6. Custom JWT-based Authentication – Using backend APIs with JSON Web Tokens for custom apps.

Types of Authentication Providers:
1. Social Providers: Google, Facebook, GitHub, Twitter, etc.
2. Enterprise Providers: Microsoft Azure AD, Okta, SAML-based systems.
3. Custom Providers: Your own backend using JWT, sessions, or OAuth.

Q21. How we achieved memoization in React?
Answer: There are a few ways memoization is used in React:
a. React.memo
b. useMemo
c. useCallback

Q22. How do you manage global state without Redux?
Answer: We manage state using React Context API, sometimes with prop drilling for simpler cases. For more scalable solutions, we use lightweight libraries like Zustand, and for persistent or session-specific data, we leverage localStorage, sessionStorage, or URL parameters.

Q23. What are the different way to share state between components?
Answer: There are several ways to share state between React components:

1. **Props (Parent → Child) ->** The simplest and most common method.Used when data needs to be passed downward in the component tree.
2. **Callback function (Child → Parent) ->** Parent passes a function to child. Allows the child to send data back or update parent state.
3. **Lifting State Up ->** Move shared state to the closest common parent. Useful when two sibling components need access to the same state.
4. **React Context API ->** Used to avoid prop drilling. Good for global state like theme, user, language. Light-weight and built-in.
5. **State Management Libraries ->** Used when the state becomes large or shared across many components.
   Common ones: Redux (most used), Zustand, Recoil, Jotai, MobX
6. **URL Params or Query Params ->** State stored in the URL. Useful for routing, filters, pagination, search keywords.
7. **Browser Storage ->** localStorage, sessionStorage when state needs to persist across refresh.
   Example: tokens, preference settings.
8. **Custom Hooks** -> Share reusable logic across components. Good for data fetching, auth logic, timers, form logic, etc.

Q24. Explain the difference between `useContext` and Redux.
Answer: useContext - useContext helps avoid prop drilling and allows global access, but it doesn't provide proper state management structure. Whenever the context value updates, all consuming components re-render, which can cause performance issues.
On the other hand, Redux, provides a predictable and scalable way to manage global state using actions and reducers. Even though we must create common actions, this structure brings consistency, better debugging, middleware support for async operations, and optimized rendering.
So, useContext is good for small, simple global state, while Redux is suitable for complex, shared and large-scale application state.

Q25. What are Redux's powerful DevTools?
Answer: Redux DevTools is a browser extension that helps developers see, track, and debug state changes in a very clear and powerful way. It gives visibility into every state update, every action, and how your state changes step-by-step.

Q26. Explain the difference between `<BrowserRouter>` and `<HashRouter>`.
Answer: BrowserRouter gives clean URLs and needs server config, while HashRouter uses a # in the URL and works without any server setup, making it good for static hosting.

Q27. What are best practices for writing maintainable React components?
Answer: Keep Components Small and Focused, Use Functional Components and Hooks, Write Reusable and Composable Components, Follow Consistent Naming Conventions, Write Clear and Readable JSX.

Q28. What is Abort controller.
Answer: AbortController is a built-in JavaScript API used to cancel asynchronous operations, most commonly fetch requests. It's useful in scenarios where a request is no longer needed.
For example, when a component unmounts in React or when you want to prevent multiple unnecessary API calls.

```
import { useEffect } from 'react';

function MyComponent() {
  useEffect(() => {
    const controller = new AbortController();

    fetch('/api/data', { signal: controller.signal })
      .then(res => res.json())
      .then(data => console.log(data))
      .catch(err => {
        if (err.name === 'AbortError') console.log('Request cancelled');
      });

    return () => controller.abort(); // abort fetch on unmount
  }, []);

  return <div>Check console for data</div>;
}
```

Why it's important:
1. Prevents memory leaks and unnecessary state updates in React.
2. Useful for optimizing network requests.
3. Cleaner and safer than using manual flags for cancelling requests.

Q29. How does the reconciliation process work in React?
Answer: When a component's state or props change then rest will compare the rendered element with previously rendered DOM and will update the actual DOM if it is needed. This process is known as reconciliation.

How it Works (Step by Step)
1. Virtual DOM Creation: When your component renders, React creates a virtual DOM — a lightweight JavaScript object representing the actual DOM.
2. Diffing Algorithm: React compares the new virtual DOM with the previous virtual DOM using the diffing algorithm. It identifies what changed — added, removed, or updated elements.
3. DOM Updates: Only the elements that changed are patched to the real DOM.

Q30. What are lazy loading and code splitting in React?
Answer: Lazy loading is a technique to load components or resources only when they are needed, instead of loading everything upfront. This reduces the initial bundle size, improving page load time and performance.

```
import React, { Suspense } from "react";

// Lazy load the component
const HeavyComponent = React.lazy(() => import("./HeavyComponent"));

function App() {
  return (
    <div>
      <h1>My App</h1>
      <Suspense fallback={<div>Loading...</div>}>
        <HeavyComponent />
      </Suspense>
    </div>
  );
}
```

Suspense: Suspense allows React to wait for something to load before rendering, showing a fallback UI meanwhile.

Key Points for Interview:
1. Used with React.lazy() or async data fetching libraries.
2. Accepts a fallback prop to show a loader or placeholder.
3. Helps in handling asynchronous rendering gracefully and improving user experience.

Q31. What is the purpose of the forwardRef function in React?
Answer: forwardRef is used to forward refs through components, allowing parent components to interact with the child's DOM node.

Q32. How does React handle forms?
Answer: React handles forms by using controlled components, where form data is controlled by the React state.

Q33. What are React portals?
Answer: React portals provide a way to render children into a DOM node that exists outside the parent component's hierarchy.

Q34. Explain the concept of suspense in React.
Answer: Suspense is a feature in React that allows components to "wait" for something before rendering, such as data fetching or code splitting.

Q35. What is Babel in React js?
Answer: Babel, is a JavaScript compiler that converts latest JavaScript like ES6, ES7 into plain old ES5 JavaScript that most browsers understand.

Q36. How can a browser read JSX file?
Answer: If you want the browser to read JSX, then that JSX file should be replaced using a JSX transformer like Babel and then send back to the browser.

Q37. How can you re-render a component without using setState() function?
Answer: You can use forceUpdate() function for re-rending any component.

Q38. Can you update props in react?
Answer: You can't update props in react js because props are read-only. Moreover, you can not modify props received from parent to child.

Q39. Explain the meaning of Mounting and Demounting
Answer: Mounting: The process of attaching the element to the DCOM is called mounting.
Demounting: The process of detaching the element from the DCOM is called the demounting process.

Q40. What is the use of 'props-types' library?
Answer: 'Prop-types' library allows us to perform runtime type checking for props and similar object in a recent application.

Q41. what is slice in React Tool Kit.
Answer: A slice in Redux Toolkit is a part of the Redux store that holds related state and logic. It includes state, reducers, and auto-generated actions in one place, making state management simpler.

Example:
========== counterSlice.js ==========
```
import { createSlice } from "@reduxjs/toolkit";

const counterSlice = createSlice({
  name: "counter",
  initialState: { value: 0 },
  reducers: {
    increment: (state) => {
      state.value += 1;
    },
    decrement: (state) => {
      state.value -= 1;
    },
    incrementByAmount: (state, action) => {
      state.value += action.payload;
    }
  }
});

export const { increment, decrement, incrementByAmount } = counterSlice.actions;
export default counterSlice.reducer;
```

========== store.js ==========
```
import { configureStore } from "@reduxjs/toolkit";
import counterReducer from "./counterSlice";

export const store = configureStore({
  reducer: {
    counter: counterReducer
  }
});
```

========== index.js ==========
```
import React from "react";
import ReactDOM from "react-dom/client";
import { Provider } from "react-redux";
import App from "./App";
import { store } from "./store";

const root = ReactDOM.createRoot(document.getElementById("root"));
root.render(
  <Provider store={store}>
    <App />
  </Provider>
);
```

========== App.js ==========

```
import React from "react";
import { useDispatch, useSelector } from "react-redux";
import { increment, decrement, incrementByAmount } from "./counterSlice";

function App() {
 const count = useSelector((state) => state.counter.value);
 const dispatch = useDispatch();

 return (
  <div>
   <h2>Count: {count}</h2>

   <button onClick={() => dispatch(increment())}>+</button>
   <button onClick={() => dispatch(decrement())}>-</button>
   <button onClick={() => dispatch(incrementByAmount(5))}>
    +5
   </button>
  </div>
 );
}

export default App;
```

==========

-> Why were hooks introduced? - To use state and lifecycle features in functional components and simplify code reuse.
-> How do you prevent unnecessary re-renders? - Use React.memo, useCallback, and useMemo.
-> Can hooks replace Redux? - For small/medium apps, useReducer + useContext can replace Redux; for large-scale state, Redux may still be useful.
-> Can two components share the same hook state? - Not directly — but they can share logic by using a custom hook.
-> What happens if you pass no dependency array to useEffect? - It will run after every render, which can impact performance.
-> What happens if you update state inside useEffect without dependencies? - It causes an infinite re-render loop, because the effect runs after every render and updates state again.
-> How does React know which state belongs to which component? - React keeps a hook call order internally — hooks must always be called in the same order on every render.
-> Can hooks be conditional? - No. Hooks must be called in the same order every render.

Note**
a. useRef ? remembers a value.
b. useMemo ? remembers a calculated results.
c. useCallback ? remembers a function.
d. React.memo ? remembers a component render.

-> Compare the hooks with lifeCycle methods:

componentDidMount -> useEffect(() => {}, [])
componentDidUpdate -> useEffect(() => {}, [deps])
componentWillUnmount -> useEffect(() => return () => { cleanup })
shouldComponentUpdate -> useMemo, useCallback

Q42. Diffrence between yarn and npm.
Answer: npm and Yarn are JavaScript package managers used to manage dependencies in Node.js applications. npm comes by default with Node.js and is widely used, while Yarn was introduced to improve speed and consistency. In recent versions, both offer similar performance and features, so the choice usually depends on project requirements and team preference. (Yarn was Created by Facebook (Meta) in 2016).

Q43. Difference between npm and npx.
Answer: 1. npm is used to install and manage packages that our project needs. Packages saved in node_modules. Locked in package.json
2. npx is used to run a tool or package once without installing it permanently. Uses latest by default unless specified.

Q44. Difference between package.json and node_module.
Answer:
package.json → The list of dependencies your project needs. Tells Node.js what packages your project requires and their versions ranges.
package-lock.json → The exact snapshot of all installed packages, including nested dependencies. So everyone gets the same setup.
node_modules → The actual folder containing all installed packages that your project uses.

NODE JS

Q1. What is Node.js?
Answer: Node.js is a JavaScript runtime environment built on Chrome's V8 engine that allows developers to run JavaScript on the server side. It is lightweight, fast, and event-driven, making it suitable for building scalable, real-time, and data-intensive applications.

Q2. Is Node.js single-threaded?
Answer: Yes, Node.js is single-threaded, but it uses an event-driven architecture to handle multiple requests efficiently without blocking, making it fast and scalable. The event loop manages asynchronous operations, while non-blocking APIs allow I/O tasks to run efficiently. For CPU-intensive tasks, Node.js can leverage worker threads so the main thread remains free.

Q3. What is the Event Loop?
Answer: The Event Loop is the core mechanism in Node.js that allows it to perform non-blocking I/O operations despite being single-threaded. It continuously monitors the call stack and the callback queue, executing asynchronous callbacks when the stack is empty. This enables Node.js to handle multiple concurrent requests efficiently without blocking the main thread.

Key points you can also mention briefly in interviews:
1. Works with callbacks, promises, and async/await.
2. Handles timers, I/O operations, and events.
3. Ensures Node.js remains fast and scalable even under heavy load.

Q4. Difference between setImmediate(), setTimeout(), and process.nextTick()?
Answer:

process.nextTick()	Executes before event loop continues
setImmediate()	Executes in the check phase
setTimeout()	Executes after specified delay

Q5. What are Streams in Node.js?
Answer: Streams in Node.js are objects that allow you to read or write data piece by piece, rather than loading the entire data into memory at once. They are especially useful for handling large files, network data, or any continuous data efficiently. Node.js provides four main types of streams: Readable, Writable, Duplex, and Transform.

Types of Streams
1. **Readable –** Data can be read from the stream (e.g., fs.createReadStream()).
2. **Writable –** Data can be written to the stream (e.g., fs.createWriteStream()).
3. **Duplex –** Data can be read and written (e.g., network sockets).
4. **Transform –** A type of duplex stream that can modify data while reading/writing (e.g., compression streams).

Q6. What is callback hell?
Answer: Callback hell happens when there are too many nested callbacks in asynchronous code, making it messy and hard to maintain. It can be avoided using Promises or async/await.

Q7. Difference between Promise and async/await?
Answer:

Promises	**async/await**
.then().catch()	Cleaner, synchronous-like
More nesting	Easier error handling
Less readable	More readable

Q8. How does Node handle async operations internally?
Answer: Node.js handles asynchronous operations using an event-driven, non-blocking architecture.

Internally, Node.js runs on a single-threaded event loop, which means it doesn't create a new thread for every request. When an async operation like a database call, file read, or API request is triggered, Node.js delegates that task to the system or a background thread pool (handled by libuv). Once the operation is completed, the result is placed in a callback queue, and the event loop picks it up and executes the corresponding callback or Promise resolution.

Because of this mechanism:
1. Node.js can handle multiple requests efficiently
2. The main thread is never blocked
3. It performs very well for I/O-heavy applications, like REST APIs

In short, Node.js uses the event loop + callbacks/promises + background threads to manage async operations efficiently.

Q9. What is Express.js?
Answer: Express.js is a minimal and flexible Node.js web application framework used to build web applications and RESTful APIs.

It provides a simple way to handle:
1. Routing (GET, POST, PUT, DELETE)
2. Middleware for request handling
3. Request and response management

Express.js sits on top of Node.js and helps reduce boilerplate code, making backend development faster and more structured.

In MERN applications, Express.js is commonly used to:
1. Build backend APIs
2. Connect frontend (React) with databases (MongoDB)
3. Handle authentication, validation, and error handling

Q10, What is Rate-limit?

Answer: Rate-limiting is a technique used in web applications and APIs to control the number of requests a user or client can make within a specific time period. It helps prevent abuse, reduce server overload, and ensure fair usage of resources. Rate-limiting can be applied per user, per IP address, or per API key, and is commonly used in combination with throttling or monitoring systems.

Example:

```
const express = require('express');
const rateLimit = require('express-rate-limit');
const app = express();

// Set a "RED limit" on requests
const limiter = rateLimit({
  windowMs: 1000, // 1 second
  max: 100,       // Max 100 requests per second
});

app.use(limiter);

app.get('/', (req, res) => {
  res.send('Hello RED!');
});
app.listen(3000);
```

Q11. What is middleware in Express?

Answer: Middleware is a function in Node.js/Express that runs **between the request and the response**. It can modify the request/response, run checks, or stop the request. Middlewares is used for logging, authentication, error handling, parsing data, validation and more.

Types of Middleware in Express

Built-in Middleware

1. **express.json() – Parses JSON data**
2. **express.urlencoded() – Parses form data**
3. **express.static() –** Serves static files

```
app.use(express.static("public"));
const express = require("express");

const app = express();
// For serve Static files to client
app.use(express.static("public"));
app.listen(3000, () => {
   console.log("Server running on port 3000");
});

// On Frontend side
// It directly serve public folder files:
http://localhost:3000/style.css
```

Third-party Middleware

1. **cors – Handles cross-origin requests**
2. **morgan – Logs requests**

```
app.use(morgan("dev"));

// Showing on logs(console)
GET /users 200 15ms
```

Custom Middleware

1. **User-defined logic (auth, validation)**

Example:

```
const authMiddleware = (req, res, next) => {
  const token = req.headers.authorization;
  if (!token) {
    return res.status(401).send("Unauthorized");
  }
  next(); // request ko aage bhejta hai
};

==============================
app.get("/dashboard", authMiddleware, (req, res) => {
  res.send("Welcome to dashboard");
});
```

Error-handling Middleware

Example: When error come, this middleware run

```
app.use((err, req, res, next) => {
  console.error(err.message);
  res.status(500).send("Something went wrong");
```

```
});
==========================
app.get("/error", (req, res, next) => {
  const error = new Error("Server error");
  next(error); // error middleware ko call Karega
});
```

Q12. Types of middleware in Express?
Answer:
1. Application-level (applies to app or specific routes)
2. Router-level (applies to specific routers)
3. Built-in (Provided by Express itself)
4. Third-party (Installed from npm)

Q13. What is next() used for?
Answer: next() passes control to the next middleware function in the request-response cycle.

Q14. How do you handle errors in Express?
Answer: Using error-handling middleware: Error-handling middleware in Express is a special type of middleware that catches and processes errors that occur during the request-response cycle. It helps prevent the application from crashing and send proper error responses to clients.

Example:
```
app.use((err, req, res, next) => {
  res.status(500).json({ error: err.message });
});
```

Q15. What is Redis?
Answer: Redis is an in-memory key-value store, that stores data in RAM, making it extremely fast. caching, fast data access, and real-time data management in applications. In Node.js applications, Redis is used to improve performance and speed.

Example:
```
const redis = require('redis');
const client = redis.createClient(); // Connect to Redis server

// Store data in Redis
client.set('username', 'Alice'); // Data is now in Redis server

// Read data from Redis
client.get('username', (err, value) => {
  console.log(value); // 'Alice' -> Node.js fetch send it to frontend
});
```

Q16. What is caching in NodeJs.
Answer: Caching is the process of temporarily storing frequently used data in a fast storage (like memory) so that future requests for the same data can be served quickly without computing or fetching it again from a slower source (like a database or API). For this we use Redis.

Q17. What is Workers in NodeJs.
Answer: Workers are helper threads in Node.js that run heavy tasks in parallel so the main thread doesn't get blocked. They make the app faster by keeping the main thread free to handle other requests.

```
// main.js
const { Worker } = require('worker_threads');

// Create a worker to do a heavy task
const worker = new Worker('./worker.js');
worker.on('message', (result) => {
  console.log('Result from worker:', result);
});

// Send data to worker
worker.postMessage(10);
```

========== **worker.js file** ==========

```
const { parentPort } = require('worker_threads');

// Worker receives data, does computation, and sends result back
parentPort.on('message', (num) => {
  const result = num * 2;
  parentPort.postMessage(result);
});
```

Q18. What is Closure?
Answer: A closure is a mechanism that allows inner function to remember the outer scope variables when it was defined, even after the outer function has returned.
The closure has three scope chains:

1. It can access its own scope means variables defined between its curly brackets ({ }).
2. It can access the outer function's variables.
3. It can access the global variables.

Example:

```
const outer = () => {
    var message = "I am outside";
    const inner = () => {
        console.log(message);  // I am outside
    };
    return inner;
};
const innerFunc = outer();
innerFunc();
```

Q19. Difference between require() and import?
Answer:
require() → CommonJS
import → ES Modules

Q20. What is Cluster?
Answer: Clustering allows the creation of multiple worker processes to handle load using all CPU cores. A master process manages these workers and distributes requests among them, improving performance and reliability.

Q19. Difference between Cluster and Worker.
Answer:

Cluster	Worker
1. Cluster is a feature/module in Node.js	Worker is a single process created by cluster
2. It manages multiple workers	It does the actual work (handles requests)
3. Uses all CPU cores	Uses one CPU core
4. Creates and controls workers	Runs the application code

In simple words:
Cluster = Manager (creates and manages workers)
Worker = Employee (handles requests)

Q20. How do you improve Node.js performance?
Answer:
1. Use async/await
2. Avoid blocking code
3. Use clustering
4. Implement caching (Redis)
5. Optimize database queries

Q21. What is REST?
Answer: REST (Representational State Transfer) is an architectural style for designing web services that use standard HTTP methods to access and manipulate resources in a stateless manner.

Key Points:
Uses HTTP methods (GET, POST, PUT, DELETE)
Works with resources (like users, products)
Is stateless (server does not store client state)

Q22. How does Node communicate with React?
Answer: Through REST APIs or GraphQL, usually using JSON over HTTP.

Q23. What is CORS?
Answer: CORS controls cross-origin requests between frontend and backend.

Q24. What is Cryptography.
Answer: Cryptography is the practice of protecting data by converting it into a secure form so that only authorized users can read or use it. It uses techniques like encryption, hashing, and digital signatures to ensure confidentiality, integrity, and authentication of data.
Type of Cryptography:

1. Encryption & Decryption
2. Hashing
3. Symmetric Cryptography: Same secret key for encrypt & decrypt
4. Asymmetric Cryptography: Two keys: Public key & Private key, encrypt by public key and decrypt by private key

Problems solved by Cryptography (in English):

- **Confidentiality** – Ensures that only authorized users can access the data.
- **Integrity** – Ensures that the data is not altered or tampered with.
- **Authentication** – Verifies the identity of the user or sender.
- **Non-repudiation** – Prevents the sender from denying that they sent the data.

Q25. How do you upload files in Node.js?
Answer: In Node.js, file uploads are commonly handled using multer middleware. It processes incoming files from client requests and saves them to the server, making the file info available in req.file or req.files.
Step 1: Install a middleware

1. npm install multer

Step 2: Set up multer in your app

```
const express = require("express");
const multer = require("multer");
const app = express();

// Set storage options
const storage = multer.diskStorage({
  destination: function (req, file, cb) {
    cb(null, "uploads/"); // folder where files will be saved
  },
  filename: function (req, file, cb) {
    cb(null, Date.now() + "-" + file.originalname); // file name
  }
});

const upload = multer({ storage: storage });
// Create uploads folder if not exists
const fs = require("fs");
if (!fs.existsSync("uploads")) {
  fs.mkdirSync("uploads");
}
```

Step 3: Create a route to upload files

```
// Single file upload

app.post("/upload", upload.single("myFile"), (req, res) => {
  res.send("File uploaded successfully: " + req.file.filename);
});

// Multiple files upload

app.post("/upload-multiple", upload.array("myFiles", 5), (req, res) => {
  res.send("Files uploaded successfully");
});
app.listen(3000, () => console.log("Server started on port 3000"));
```

Step 4: Client-side (HTML form)

```
<form action="/upload" method="POST" enctype="multipart/form-data">
  <input type="file" name="myFile" />
  <button type="submit">Upload</button>
</form>
```

Q26. How do you deploy a Node.js app?
Answer: To deploy a Node.js app, we upload the code to a server, install dependencies, and start the application. Tools like PM2 are used to keep the app running in production.

Q27. What is SDLC.
Answer: SDLC (Software Development Life Cycle) is a structured process used to develop high-quality software. It includes requirement analysis, feasibility study, design, coding, testing, deployment, and maintenance. Requirements are gathered by BA/PA, then the system is designed, developed, tested, and deployed. Post-deployment, maintenance is provided to fix issues and improve the application.

Q28. SDLC – Software Development Life Cycle (Phases)
Answer: The Software Development Life Cycle (SDLC) is a structured process followed to design, develop, test, and maintain software applications.

1. Requirement Analysis: In a project-based company, the Project Analyst (PA) gathers requirements, while in a service-based company, the Business Analyst (BA) is responsible for collecting and documenting client requirements.
2. Feasibility Study: This phase involves stakeholders such as PA/BA, HR, Technical Architect, and Finance. It includes cost analysis, risk assessment, technical feasibility, and resource planning to ensure the project is viable.
3. Design: The system architecture and detailed design are created, including database design, system flow, and technology stack selection.
4. Coding (Development): Developers convert the approved design into actual software by writing clean, efficient, and scalable code.
5. Testing: The software is tested to ensure it works as expected, meets requirements, and is free from defects.
6. Deployment: After successful testing, the application is deployed to the production environment for end users.
7. Maintenance: Post-deployment, maintenance and support are provided to fix issues, improve performance, and add enhancements for a specified period.

Q29. What is Difference between GET and POST API.

Answer: GET and POST are HTTP methods used for client–server communication. GET is mainly used to fetch or read data from the server, where the data is sent through URL query parameters and is visible in the browser address bar. Due to URL length limitations, the amount of data sent using GET is restricted. GET requests are less secure because the data is exposed in the URL, can be cached by browsers, and can be bookmarked. GET is an idempotent method, meaning it does not change the server's state.

On the other hand, POST is used to send or create data on the server. The data is sent in the request body, making it invisible in the browser URL and allowing much larger payloads. POST requests are more secure compared to GET since the data is not exposed in the URL. They are not cached by default, cannot be bookmarked, and are non-idempotent, as they usually modify or create data on the server.

Idempotent: Repeating the same operation does not change the outcome after the first time.

Q30. What is an LRU (Least Recently Used) Cache?
Answer: An LRU Cache is a data structure that stores a limited number of items and removes the least recently used item when the cache reaches its limit.

Q31. Deep copy and Shallow copy.
Answer:
const obj = { name: "Alice", address: { city: "NY" } };
Shallow Copy:
1. A shallow copy copies only the top-level properties of an object or array.
2. Nested objects/arrays are still referenced, not cloned.
3. Changing a nested object in the copy also affects the original.

```
// Shallow clone
const shallow = { ...obj };
shallow.address.city = "LA";
console.log(obj.address.city); // "LA" → original affected
```

Deep Copy:
1. A deep copy recursively copies all levels of an object or array.
2. Nested objects are fully cloned, so changes in the copy do not affect the original.

```
// Deep clone
const deep = JSON.parse(JSON.stringify(obj));
deep.address.city = "Chicago";
console.log(obj.address.city); // "LA" → original not affected
```

Q32. Flatten nexted array.
Answer:

```
// Using flat() method
const arr = [1, [2, [3, 4], 5], 6];
console.log(arr.flat());  // Output: [1, 2, [3, 4], 5, 6]  → flattened by 1 level

console.log(arr.flat(2));   // Output: [1, 2, 3, 4, 5, 6] → flattened 2 levels

console.log(arr.flat(Infinity));  // Output: [1, 2, 3, 4, 5, 6] → completely flatten any depth

============================================================
//using recursion
const flatten = arr => arr.reduce(
  (acc, val) => acc.concat(Array.isArray(val) ? flatten(val) : val),
  []
);

============================================================
// Using Normal code
const flatten = arr => {
  let result = [];
  for (let i = 0; i < arr.length; i++) {
    if (Array.isArray(arr[i])) {
      // if it's an array, flatten it and add to result
      result = result.concat(flatten(arr[i]));
    } else {
      // if it's not an array, just add it
      result.push(arr[i]);
    }
  }
  return result;
};

// Example:
console.log(flatten([1, [2, [3, 4], 5], 6]));   // Output: [1, 2, 3, 4, 5, 6]
```

Authentication & Security

Q1. What is JWT?
Answer: JWT (JSON Web Token) is a secure token used in web applications for authentication and authorization. It allows servers to verify user identity without storing session data, as all necessary information is self-contained and signed within the token.
Strucure of JWT - A JWT consists of three parts, separated by dots (.): header.payload.signature

Ex- eyJhbGciOiJIUzI1NiIsInR5cCI9.eyJzdWIiOiIgRG9lIiwiYWRtaW4iOnRydWUsImlhdCI6MTUxNjIzOTAyMn0.KMUFsIDM6H9FNFUROf3wh7SmqJp-QV30
1. Header – Contains the token type (JWT) and signing algorithm (e.g., HS256).
2. Payload – Contains the user data or claims (e.g., user ID, role).
3. Signature – Created by combining the header, payload, and a secret key; used to verify the token's integrity.

Example:
```
const jwt = require('jsonwebtoken');
const payload = { userId: '12345' };
const secret =process.env.secret_key;   //Create it mannualy like: $2b$10$EixZaYVK1fsbw1ZfbX3OXePaWxn96p36YfGz2r9/YyqQ8f7sO9jW this.

// Generate JWT
const token = jwt.sign(payload, secret, { expiresIn: '1h' });
console.log('JWT Token:', token);

// Verify JWT
const decoded = jwt.verify(token, secret);
console.log('Decoded Payload:', decoded);
```

Q2. How do you secure APIs in Node.js?
Answer: Node.js APIs are secured using authentication (JWT), rate limiting, input validation, password hashing, HTTPS, security headers, and proper error handling to protect data and prevent unauthorized access.

1. Authentication & Authorization: Use JWT (JSON Web Tokens) or OAuth to verify user identity and control access to resources.
2. Rate Limiting: Limit the number of requests per user/IP to prevent abuse and DDoS attacks (e.g., express-rate-limit).
3. Input Validation & Sanitization: Validate request data using libraries like Joi or express-validator to prevent invalid or malicious input.
4. Password Hashing: Hash passwords using bcrypt before storing them in the database.
5. Use HTTPS: Encrypt data in transit using SSL/TLS to protect sensitive information.
6. Security Headers: Use Helmet to set secure HTTP headers and protect against common web vulnerabilities.
7. CORS Configuration: Restrict API access to trusted domains using CORS.
8. Proper Error Handling & Logging: Avoid exposing sensitive information in error messages and log errors securely.

Q3. What is Helmet?
Answer: Helmet is an Express.js middleware that secures applications by setting safe HTTP headers, helping protect against common security threats.

Why Use Helmet?
1. Improves API and application security
2. Protects against common web attacks
3. Easy to use with minimal configuration

Helmet configures multiple security-related headers, such as:
1. Content-Security-Policy (CSP) → Prevents XSS attacks
2. X-Frame-Options → Protects against clickjacking
3. X-Content-Type-Options → Prevents MIME-type sniffing
4. Strict-Transport-Security (HSTS) → Enforces HTTPS
5. Referrer-Policy → Controls referrer information

Example:
```
const express = require('express');
const helmet = require('helmet');

const app = express();
app.use(helmet());

app.listen(3000, () => {
    console.log('Server secured with Helmet');
});
```

Q4. What is Data leakage, how find it?
Answer: Data leakage is found by reviewing API responses, logs, error messages, and access controls to ensure sensitive data is not exposed.
Security testing tools and monitoring network traffic also help detect unintended data exposure.

Q5. What is Node.js Profilers?
Answer: Node.js profilers are tools that help monitor and analyze application performance by tracking CPU, memory, and execution time.
They are used to identify performance issues and optimize Node.js applications.

What Node Profilers Do
1. Measure CPU usage
2. Track memory consumption

3. Detect slow functions
4. Find performance bottlenecks
5. Help identify memory leaks

Common Node.js Profilers
1. Node.js built-in profiler (--inspect, --prof): node --inspect app.js
2. Chrome DevTools
3. Clinic.js: It is a Node.js tool used to profile and debug performance problems like high CPU usage, event loop blocking, and memory leaks.
4. PM2 monitoring
5. New Relic / Datadog - Datadog is a cloud-based monitoring and analytics platform that helps developers and operations teams monitor applications, infrastructure, and logs in real time.

Q6. Is JWT stateful or stateless?
Answer: JWT is stateless because the server does not store session information. All required data is present in the token, which is verified on each request.
Note: JWT can be made stateful if stored in a database or blacklist, but by default it is stateless.

Q7. Give Difference beetween PUT and PATCH.
Answer: PUT is used to replace the entire resource, while PATCH updates only the provided fields. PUT requires sending full data, whereas PATCH sends partial data.

Q8. Name all HTTP methods.
Answer: Comman HTTP Methods (Request Types) are:

GET – Get data from server
POST – Send new data to server
PUT – Update/replace full data
PATCH – Update partial data
DELETE – Delete data
HEAD – Same as GET but without response body
OPTIONS – Get allowed methods / CORS info
TRACE – Diagnostic testing
CONNECT – Create tunnel (used in HTTPS)

Q9. How do you handle environment variables?
Answer: Using dotenv:
require('dotenv').config();

Q10. HTTP Status Codes List
1xx – Informational
100 Continue
101 Switching Protocols
102 Processing

2xx – Success
200 OK
201 Created
202 Accepted
204 No Content
206 Partial Content

3xx – Redirection
301 Moved Permanently
302 Found
304 Not Modified
307 Temporary Redirect
308 Permanent Redirect

4xx – Client Error
400 Bad Request
401 Unauthorized
403 Forbidden
404 Not Found
405 Method Not Allowed
408 Request Timeout
409 Conflict
410 Gone
422 Unprocessable Entity
429 Too Many Requests

5xx – Server Error
500 Internal Server Error
501 Not Implemented
502 Bad Gateway
503 Service Unavailable
504 Gateway Timeout

AWS

Q1. What is AWS, how it works.
Answer: AWS is commonly used to deploy MERN applications. For example, EC2 is used to host Node.js APIs, S3 and CloudFront serve React apps and images, Auto Scaling handles traffic spikes, and CloudWatch is used for monitoring. AWS helps companies scale securely and cost-effectively.

AWS (Amazon Web Services) is a cloud computing platform provided by Amazon that offers on-demand services like servers, storage, databases, networking, and deployment tools. It allows companies to build, deploy, and scale applications without managing physical servers, and they only pay for what they use. For a MERN stack app, AWS can be used to deploy the React frontend, host Node.js/Express backend, store data in databases, manage files, and handle scalability and security.

Q2. Why companies use AWS?
Answer:

1. Cost-effective (pay-as-you-go)
2. Scalable
3. durable storage
4. Global infrastructure
5. Secure access

Q3. Have you used AWS?
Answer: Yes, I have basic experience using AWS services like EC2 for hosting Node.js applications, S3 for static files, and CloudWatch for monitoring.

Q4. Why AWS?
Answer:

1. Easy deployment
2. Scales when users increase
3. No physical servers needed
4. Load Balancer → Distributes traffic evenly
5. EC2 → Runs multiple backend servers
6. S3 → Stores uploaded images
7. IAM → Controls access permissions

Q5. Why is AWS used in MERN applications?
Answer: AWS is used in MERN applications for scalability, cost efficiency, security, and high availability. It allows hosting React frontend, Node.js backend, handling file storage, monitoring, and scaling traffic automatically.

Q6. Which AWS services are commonly used in MERN stack?
Answer:

1. EC2 – Hosting Node.js / Express backend
2. S3 – Storing React build files and images
3. CloudFront – CDN (Content Delivery Network) for fast frontend delivery
4. Load Balancer – Distributes traffic
5. CloudWatch – Logs and monitoring
6. IAM – Security and access management
7. MongoDB Atlas (on AWS) – Database

Q7. How do you deploy a MERN application on AWS?
Answer:

1. Build React app and upload it to S3
2. Use CloudFront to serve frontend (CloudFront give url and if we hit this url our React app render.)
3. Deploy Node.js backend on EC2 or Elastic Beanstalk
4. Connect MongoDB using MongoDB Atlas
5. Use Nginx as reverse proxy
6. Enable monitoring using CloudWatch

Q8. What is EC2?
Answer: EC2 (Elastic Compute Cloud) is a virtual server in AWS used to run backend applications like Node.js APIs. It provides full control over OS, CPU, memory, and scaling.

Q9. What is S3?
Answer: S3 (Simple Storage Service) is an object storage service used to store static files like images, videos, and frontend build files. It is highly durable and cost-effective.

Q10. Difference between EC2 and S3?
Answer:

EC2	S3
Virtual server	Storage service
Runs backend code	Stores static files
Used for APIs	Used for images, frontend
Requires OS management	Fully managed

Q11. What is IAM?
Answer: IAM (Identity and Access Management) controls who can access AWS resources. It uses users, roles, and policies to provide secure, role-based access.

Q12. What is CloudWatch?
Answer: CloudWatch is a monitoring service that tracks server metrics, application logs, and sets alerts when issues occur like high CPU usage or server downtime.

Q13. What is Elastic Beanstalk?
Answer: Elastic Beanstalk is a PaaS service that simplifies deployment. Developers just upload code, and AWS handles servers, scaling, and monitoring automatically.

Q14. How do you secure AWS applications?
Answer:

1. Use IAM roles
2. Enable Security Groups
3. Use HTTPS (SSL)
4. Store secrets in Environment Variables
5. Restrict public access to S3

Q15. Have you used AWS?
Answer: Yes, I have used AWS for deploying Node.js applications on EC2, storing files in S3, and monitoring using CloudWatch. I am continuously improving my AWS knowledge.

Q16. What is the difference between CloudFront and S3?
Answer: Amazon S3 is a storage service that stores data like images, videos, and files. CloudFront is a CDN(Content Delivery Network) that delivers this data to users quickly by caching it near them. This reduces latency and improves performance.

Note** S3 stores data, CloudFront delivers it fast.
Note** Latency is the time taken between sending a request and receiving a response. CloudFront reduces latency and improves performance by caching data closer to the user.

Q17. AWS vs Traditional Hosting?
Answer: AWS offers scalability, pay-as-you-go pricing, high availability, and security, while traditional hosting has fixed resources and limited scalability.

Q18. d/b high level design and low level design.
Answer:
High Level Design (HLD): HLD gives an overall system view, including system architecture, major modules, data flow, and technology stack. It focuses on what the system will do, not on coding or internal logic.
Low-Level Design (LLD): LLD explains the detailed internal working of each module, including class diagrams, APIs, logic, and workflows.
It focuses on how the system will be implemented at the code level.

Q19. What is MonoLithic and MicroServices?
Answer:
Monolithic Architecture: In a monolithic architecture, the entire application (UI, business logic, and database) is built as a single unit. All modules are tightly coupled and deployed together, making it simple to develop but harder to scale and maintain.
Microservices Architecture: In microservices, the application is divided into small, independent services. Each service handles a specific function, can be developed, deployed, and scaled independently, improving flexibility and scalability.

Q20. What you understand for System architecture.
Answer: System architecture refers to the conceptual model that defines the structure, behavior, and more views of a system. It outlines how different components of a system, such as hardware, software, and networks, interact with each other to achieve the overall functionality and goals of the system. Essentially, it's like the blueprint of a system, helping teams design, develop, and maintain it efficiently.

Jenkins

Q1. What is Jenkins?
Answer: Jenkins is an open-source automation tool used for continuous integration (CI) and continuous delivery (CD). It helps developers automatically build, test, and deploy applications whenever code changes, reducing manual effort and errors.

Q2. Why is Jenkins used in MERN projects?
Answer: In MERN projects, Jenkins is used to:

1. Automatically build Node.js backend and React frontend
2. Run unit tests and integration tests
3. Deploy applications to AWS (EC2, S3, Elastic Beanstalk)
4. Ensure faster and error-free releases

Q3. How does Jenkins work?
Answer:

1. Developer pushes code to GitHub/GitLab
2. Jenkins detects the change (via webhook or polling)
3. Jenkins triggers a build pipeline
4. Pipeline runs tests, builds app, and deploys it automatically

Q4. Jenkins vs Manual Deployment.

Jenkins	**Manual Deployment**
Automated builds and deployment	**Done manually each time**
Reduces human errors	**Prone to errors**
Fast CI/CD	**Time-consuming**
Easy rollback and monitoring	**Harder to track issues**

Q5. What is the Key Jenkins Components.
Answer:

Jenkins Server – Runs Jenkins and manages pipelines
Jobs / Pipelines – Defines tasks (build, test, deploy)
Plugins – Add extra functionality (GitHub, NodeJS, Docker)
Nodes / Agents – Machines that run jobs

Q6. Real-World Example for MERN Stack
Scenario: Deploying a MERN application automatically to AWS.
How Jenkins is used:

1. GitHub webhook triggers Jenkins pipeline on code push
2. Node.js backend is built and tested
3. React frontend is built
4. Artifacts are deployed to EC2 or S3
5. CloudWatch monitors deployment success
6. Rollback is automated in case of failure

Benefit:

1. CI/CD reduces deployment time from hours to minutes
2. Ensures consistency across environments

Q7. What is a Jenkins Pipeline?
Answer: A pipeline is a sequence of automated steps (build → test → deploy). It can be written in Jenkinsfile using Declarative or Scripted syntax.
Example for MERN:

```
pipeline {
  agent any
  tools: {nodejs "NODEJS"}
  stages {
    stage('Build Frontend') {
      steps {
        sh 'cd frontend && npm install && npm run build'
      }
    }
    stage('Build Backend') {
      steps {
        sh 'cd backend && npm install'
      }
    }
    stage('Deploy') {
      steps {
        sh 'scp -r backend ec2-user@server:/var/www/app'
      }
    }
  }
}
```

Q8. Why Jenkins is popular?

Answer: 1. Open-source and free
2. Highly customizable with plugins
3. Works with any language (Node.js, Java, Python, etc.)
4. Integrates with Git, Docker, AWS, and Kubernetes

Q9. Have you used Jenkins? (HR Answer)

Answer: "Yes, I have used Jenkins to automate builds and deployments of MERN applications. For example, I created pipelines that build React and Node.js code, run tests, and deploy the application to AWS EC2 and S3. It helped reduce deployment errors and improve delivery speed."

Q10. Latest version and What language we use to write Jenkins pipeline.

Answer: The latest overall stable Jenkins release is 2.545 (released Jan 6, 2026). Jenkins Pipeline scripts (usually in a file called Jenkinsfile) are written in Groovy language.

LA Pipeline :

```
pipeline {
    agent any
    tools {nodejs "NODEJS"}
    stages {
        stage('Process Completed') {
          steps {
            sh '''
            '''
            echo "Git Pull * Pm2 Restarted successfully===========>"
                }
        }
        stage('Git Pull For Admin-Server') {
          steps {
            sshPublisher(publishers:
            [sshPublisherDesc(configName: 'Prod-LA-Super-Admin-Server',
            transfers: [sshTransfer(
              cleanRemote: false, excludes: '',
              execCommand: 'cd /home/ec2-user/LA-Portal/certifyme-portal;
              git pull https://teamsf:appPassword@bitbucket.org/certifyme-portal/certifyme-portal.git LA-BACKEND-PRODUCTION; npm install; pm2 restart bin/www --name="SuperAdmin-BE"',
              execTimeout: 120000, flatten: false, makeEmptyDirs: false, noDefaultExcludes: false,
              patternSeparator: '[, ]+', remoteDirectory:'',
              remoteDirectorySDF: false, removePrefix: '', sourceFiles: ''
            )],
            usePromotionTimestamp: false, useWorkspaceInPromotion: false, verbose: true)])
          }
        }
        stage('Git Pull For Agent-Server') {
          steps {
            sshPublisher(publishers:
            [sshPublisherDesc(configName: 'Prod-LA-Super-Admin-Server',
            transfers: [sshTransfer(
              cleanRemote: false, excludes: '',
              execCommand: 'cd /home/ec2-user/AdminPortal/certifyme-portal;
              git pull https://teamsf:appPassword@bitbucket.org/certifyme-portal/certifyme-portal.git LA-BACKEND-PRODUCTION; npm install; pm2 restart bin/www --name="Leasing-BE"',
              execTimeout: 120000, flatten: false, makeEmptyDirs: false, noDefaultExcludes: false,
              patternSeparator: '[, ]+', remoteDirectory:'',
              remoteDirectorySDF: false, removePrefix: '', sourceFiles: ''
            )],
            usePromotionTimestamp: false, useWorkspaceInPromotion: false, verbose: true)])
          }
        }
    }
    post {
      success {
        mail bcc: 'himanshunavgotri@gmail.com',
        body: "Check console output at $BUILD_URL to view the results. Project: ${env.JOB_NAME} Build Number: ${env.BUILD_NUMBER}",
        cc: '',
        from: '',
        replyTo: '',
        subject: 'Beta Site Phase 2 - pm2 restarted successfully',
        to: 'himanshunavgotri@gmail.com'
      }
      failure {
        emailext body: 'Check console output at $BUILD_URL to view the results. \n\n ${CHANGES} \n\n ---------------------------- \n${BUILD_LOG, maxLines=100, escapeHtml=false}',
        to: "himanshunavgotri@gmail.com",
        subject: 'LA-Admin Phase 2 - pm2 restarted Failed'
      }
      unstable {
        mail bcc: 'himanshunavgotri@gmail.com', body: "<b>Check console output at $BUILD_URL to view the results</b><br>Project: ${env.JOB_NAME} <br>
        Build Number: ${env.BUILD_NUMBER} <br> URL build: ${env.BUILD_URL}", cc: '', charset: 'UTF-8', from: '', mimeType: 'text/html',
        replyTo: '',
        subject: " LA-Admin Phase 2 - pm2 Unstable",
        to: "teamcertifyme@blucursor.com";
      }
    }
}
```

Docker

Q1. What is Docker?
Answer: Docker is a platform that allows developers to package applications and their dependencies into containers, ensuring the app runs consistently across different environments.

Q2. Why is Docker used in MERN projects?
Answer: In MERN projects, Docker helps to:

1. Ensure Node.js backend and React frontend run the same in development, testing, and production
2. Avoid "it works on my machine" issues
3. Simplify deployment to AWS, Jenkins, or Kubernetes
4. Quickly scale applications

Q3. How Docker works?
Answer:

1. Create a Dockerfile defining the app environment
2. Build a Docker image from the Dockerfile
3. Run a Docker container from the image
4. Containers can be deployed anywhere (local, cloud, CI/CD pipeline)

Q4. Key Docker Components:

Component	Description
Dockerfile	Instructions to build the image
Image	Read-only template of the app
Container	Running instance of an image
Docker Hub	Repository to store images
Volumes	Persist data outside containers
Networks	Connect multiple containers

Q5. Have you used Docker? (HR Answer)
Answer: Yes, I have used Docker to containerize MERN applications. For example, I created Docker images for Node.js backend and React frontend, used docker-compose to run MongoDB and the full stack together, and deployed the containers to AWS. It ensured faster and consistent deployments.

MongoDB

Q1. Difference between MongoDB and Mongoose?
Answer:
MongoDB : MongoDB is a NoSQL database for storing data in collections and documents.
Mongoose: Mongoose is a Node.js library that works with MongoDB, providing schemas, validation, and easy database operations.

Q2. What are Mongoose hooks?
Answer: Hooks (middleware) are functions that run before or after operations like: save, find and remove.
Example: hashing passwords before save.

```
const mongoose = require('mongoose');
const bcrypt = require('bcrypt');

const userSchema = new mongoose.Schema({
  username: String,
  password: String
});

// Pre-save hook to hash password

userSchema.pre('save', async function(next) {
  if (this.isModified('password')) { // Only hash if password is new or changed
    const salt = await bcrypt.genSalt(10);
    this.password = await bcrypt.hash(this.password, salt);
  }
  next();
});

// Post-save hook
// userSchema.post('save', function(doc) {
//    console.log('After saving:', doc.name);
// });

const User = mongoose.model('User', userSchema);

// Creating a new user

const createUser = async () => {
  const user = new User({ username: 'Alice', password: 'mypassword123' });
  await user.save();
  console.log('User saved with hashed password:', user.password);
};

createUser();
```

O/P: User saved with hashed password: $2b$10$EixZaYVK1fsbw1ZfbX3OXePaWxn96p36YfGz2r9/YyqQ8f7sO9jW.

Q3. What is indexing in MongoDB?
Answer: Indexing is a technique used in MongoDB to improve the speed of data retrieval. It works like an index in a book—instead of scanning every document in a collection, MongoDB can quickly locate the required data using the index.

Key Points:

1. Indexes make queries faster but can slightly slow down write operations (insert, update, delete).
2. MongoDB supports different types of indexes:
   a. Single Field Index – on one field
   b. Compound Index – on multiple fields
   c. Unique Index – ensures field values are unique
   d. Text Index – for text search
   e. TTL Index – automatically deletes documents after a certain time

Q4. What is DB optimization?
Answer: Database optimization is the process of improving query performance, reducing response time, and efficiently managing resources to make applications faster and scalable.

Common DB Optimization Techniques:

1. **Indexing:** Create indexes on frequently queried fields to speed up searches. **Example:** userSchema.index({ email: 1 }) in MongoDB.
2. **Query Optimization:** Avoid full collection scans; fetch only required fields using .select(). Use .limit(), .skip(), and filters to reduce load.
3. **Schema Design:** Design normalized or denormalized schemas depending on query patterns. Avoid unnecessary joins or deep nesting.
4. **Caching:** Use Redis or in-memory caches for frequently accessed data.
5. **Connection Pooling:** Reuse database connections instead of creating new ones for each request.
6. **Archiving Old Data:** Move rarely accessed data to another collection or database.
7. **Monitoring & Profiling:** Track slow queries using MongoDB profiler, Datadog, or other APM tools.

Q5. What are database-level constraints?
Answer: Database-level constraints are rules enforced by the database to maintain data integrity. They prevent invalid or inconsistent data, such as duplicates or null values, from being stored.

Common Database Constraints
1. PRIMARY KEY – Uniquely identifies each record
2. FOREIGN KEY – Maintains relationship between tables
3. UNIQUE – Ensures values are not duplicated
4. NOT NULL – Prevents empty values
5. CHECK – Validates a condition
6. **DEFAULT –** Sets a default value

Q5. Basic String Methods:
Answer:
length Returns string length
charAt(index) Character at given index
charCodeAt(index) Unicode of character
at(index) Character (supports negative index)

🔹 **Case Conversion**
toUpperCase() Converts to uppercase
toLowerCase() Converts to lowercase
toLocaleUpperCase() Locale-based uppercase
toLocaleLowerCase() Locale-based lowercase

🔹 **Searching Methods**
includes(str) Checks if substring exists
indexOf(str) First index of substring
lastIndexOf(str) Last index of substring
startsWith(str) Checks start
endsWith(str) Checks end
search(regex) Searches using regex
match(regex) Matches regex
matchAll(regex) Returns all matches

🔹 **Extracting / Slicing**
slice(start, end) Extracts part of string
substring(start, end) Similar to slice (no negative index)
substr(start, length) ⚠️ Deprecated

🔹 **Replace & Modify**
replace(old, new) Replaces first match
replaceAll(old, new) Replaces all matches
concat(str) Joins strings
repeat(count) Repeats string

🔹 **Trimming (Remove spaces)**
trim() Removes both-side spaces
trimStart() Removes leading spaces
trimEnd() Removes trailing spaces

🔹 **Splitting & Joining**
split(separator) Converts string to array // separator == splite(“ ”)

🔹 **Padding**
padStart(length, char) Pads at start
padEnd(length, char) Pads at end

🔹 **Conversion**
toString() Converts to string
valueOf() Returns primitive value

🔹 **Example**
```
let str = " Hello JavaScript ";

str.trim();          // "Hello JavaScript"
str.toUpperCase();   // " HELLO JAVASCRIPT "
str.includes("Java"); // true
str.split(" ");      // ["Hello", "JavaScript"]
```

Some important cases:
[100] == 100 //true || '100' == 100 //true || [] == 0 //true || '' == false // true || [100] === 100 //false || '100' === 100 //false || [] === 0 // false '' === false // false || '' == 0 // true || false == 0 //true || null == undefined //true || '' === 0 //false || false === 0 //false || null === undefined // false

Operation / Method	Description	Example
Create Object	Define a new object.	let obj = { name: "Himanshu", age: 25 };
Access (Dot Notation)	Get property value using dot .	obj.name // "Himanshu"
Access (Bracket Notation)	Get property value using [], works with dynamic keys.	obj["age"] // 25
Add Property	Add a new key-value pair.	obj.city = "Pune";
Update Property	Change existing value.	obj.age = 26;
Delete Property	Remove a property completely.	delete obj.city;
Check Property (in)	Check if property exists in object or prototype chain.	"age" in obj // true
Check Property (hasOwnProperty)	Check if property exists only in object itself.	obj.hasOwnProperty("age") // true
Object.keys()	Returns array of property names (keys).	Object.keys(obj) // ["name","age"]
Object.values()	Returns array of values.	Object.values(obj) // ["Himanshu",26]
Object.entries()	Returns array of [key, value] pairs.	Object.entries(obj)
for…in loop	Loop through keys of object.	for (let k in obj) console.log(k,obj[k]);
forEach on entries	Loop modern way using Object.entries().	Object.entries(obj).forEach(([k,v])=>console.log(k,v));
Spread (...)	Clone or merge objects.	let copy = { ...obj };
Object.assign()	Copy/merge objects.	Object.assign({}, obj, { city:"Pune" })
JSON.stringify()	Convert object → JSON string.	JSON.stringify(obj)
JSON.parse()	Convert JSON string → object.	JSON.parse('{"name":"Himanshu"}')
Object.freeze()	Prevent add/update/delete (read-only).	Object.freeze(obj);
Object.seal()	Prevent add/delete, allow update.	Object.seal(obj);
Object.getOwnPropertyDescriptor()	Get metadata about a property.	Object.getOwnPropertyDescriptor(obj,"name")
Object.defineProperty()	Add/modify property with custom rules.	Object.defineProperty(obj,"id",{value:101,writable:false});
Destructuring	Extract values into variables.	let { name, age } = obj;
Optional chaining ?.	Avoid error if property doesn't exist.	obj.address?.city // undefined
Default value in destructuring	Provide fallback if missing.	let { country="India" } = obj;

Code:
1. Remove duplicate:
const numbers = [1, 2, 4, 5, 2, 4, 9, 11, 4, 11];
const colors = ["red", "pink", "red", "blue", "black", "pink"];

Const uniqueNumbers = [...new Set(numbers)]; **// [1, 2, 4, 5, 9, 11]**
Const uniqueColors = [...new Set(colors)]; **// ['red', 'pink', 'blue', 'black']**

2. Remove falsy Value:
const falsyArray = [7, null, 10, false, NaN, 'Raj', '', 'Jay', undefined, 0]
const nonFalsy = falsyArray.filter(Boolean); **// [7, 10, 'Raj', 'Jay'];**

3. Readable Numbers:
const largeNumber = 45_000_000_000;
console.log(largeNumber === 45000000000; **// true**
const largeNumber = 45e9;
console.log(largeNumber === 45000000000; **// true**

4. Object Destruction on Array:

```
const colorCodes = [“#FFFFFF”, “#000000”, “#FF0000”, “#808080”, “#FFFF00”];
const { 0: firstColor, 4: lastColor } = colorCodes;

console.log(firstColor);    // #FFFFFF
console.log(lastColor);    // #FFFF00
```

5. Skip values in array Destructuring:

```
const scores = [50, 40, 30, 80, 90]
const [, , …restScores] = scores;
console.log(restScores);   //[30, 80, 90]
```

6. Filter with JSON.Stringify

```
const employee = {
    id: 1,
    name: “Raj”,
    address: {
        city: “Surat”
        state: “Gujrat”
        country: “India”
    }
}
const filterEmployee = JSON.stringify(employee, filters);
console.log(filterEmployee);   // {"name":"Raj","address":{"city":"Surat","country":"India"}}
```

7. Convert to a flat array using Array.flat

```
const numbers = [1, 2, [3,4], [5,[6,7]]];
const flatWithoutDepth = numbers.flat();  // [ 1, 2, 3, 4, 5, [ 6, 7 ] ]
const flatDepth1 = numbers.flat(1);  // [ 1, 2, 3, 4, 5, [ 6, 7 ] ]
const flatDepth2 = numbers.flat(2);  // [ 1, 2, 3, 4, 5, 6, 7 ]
```

8. Lock an Object using Object.freeze

```
const employee = {
   id: 1,
   name: "Jhon"
};

Object. freeze(employee);
employee.name = "Rob";// Throws an error in strict mode
console. log(employee.name); // Jhon
```

9. ADD DYNAMIC PROPERTY TO OBJECT

```
const dynamicProperty = 'age';
const employee = { [dynamicProperty]: 28 };
console. log(employee); // { age: 28 }
const person = { [`${dynamicProperty}Value']: 28 };
console.log(person); // { ageValue: 28 }
```

10. Create an object from key-value pairs using: The Object.fromEntries() method transforms a list of key-value pairs into an object.

```
const videoEntries = [
["id", 1],
["title", "Video-1"],
["size", "505 MB"],
["active", true]
];

const video = Object. fromEntries(videoEntries);
console. log(video);  // { id: 1, title: 'Video-1', size: '505 MB', active: true }
```

11. Tests every element of the array using Array.every

```
const employees = [
{ id: 1, name: "Alen", active: true },
{ id: 2, name: "Jhon", active: false },
{ id: 3, name: "Rob", active: true }
];
const isAllActive = employees.every( (employee) => employee.active === true);  //False
const isAllActive = employees.some( (employee) => employee.active === true);  //False
```

12. Mask numbers using slice and padStart

```
const cardNumber = "8844663344221199";
const last4Digit = cardNumber.slice(-4);
```

```
const maskNumber = last4Digit.padStart(cardNumber.length, "*");
console. log(maskNumber); // ********** 1199
```

13. String to a number

```
const code = '440';
console.log(+code); // 440
console. log(typeof +code); // number
```

14. How to remove duplicates from an array?

```
let unique = [...new Set([1,2,2,3])]; // [1,2,3]
```

15. How to generate random numbers in JS?

using Math.random()

```
let num = Math.floor(Math.random() * 10); // 0 to 9
```

16. How to merge two arrays?

Use spread or concat.

```
let merged = [...arr1, ...arr2];
```

17. What is a constructor function?

A function used to create instances using new.

```
function Car(make) {
  this.make = make;
}
let honda = new Car('Honda');
```

18. What is a default case in switch?

Executed if no case matches.

19. What is recursion?

A function calling itself.

```
function fact(n) {
  if (n <= 1)
    return 1;
  return n * fact(n - 1);
}
```

20. How to compare two arrays?

Use every() and length.

```
function equal(a, b) {
  return a.length === b.length && a.every((v, i) => v === b[i]);
}
```

Introduction

Q1. Introduction to HR
Answer: I am a MERN Stack Developer with 3 years of experience in building scalable and reliable web applications. In my current organization, I have worked on three major projects, contributing across both frontend and backend development.

Currently, I am working on a US-based real estate project focused on housing rental solutions. In this project, I handle end-to-end development using React and Node.js, and I extensively work with MongoDB Aggregation Pipelines to optimized queries that improved reporting, analytics, and overall application performance..I have also integrated third-party services such as PandaDoc for document automation and Stripe for payment processing.

Previously, I worked on an internal HRMS project designed to manage and track employee information. In this project, I was responsible for developing and maintaining both the frontend and backend modules.

Additionally, I have contributed to an AI-based face recognition project, where I integrated data from a Python-based API, stored the processed data in the database, and rendered it on the UI using React.
Overall, these experiences have strengthened my full-stack development skills and provided strong exposure to real-world, production-level applications.

Q2. Why are you leaving your current company?
Answer: I'm looking to focus on my career growth and learning new technologies by working on various projects. I'm grateful for everything I've learned at my current company, but I'm now seeking new challenges where I can further expand my skills and working on new technologies.

Q3. Why are you relocate to this city?
Answer: I'm interested in relocating to this city because it offers excellent opportunities in my field and aligns with my career goals. I'm excited about the prospect of contributing to your organization while also experiencing the professional and personal growth that comes with living and working in a new environment.

Q4. Regarding the work culture and policies.
Answer: bluCursor offers a supportive and collaborative work environment. Overall, the organization maintains a healthy work–life balance. The company policies are well-defined and structured. Details regarding leave, working hours, and performance evaluations are communicated clearly and transparently. Overall, it has been a good learning experience for me, especially in terms of skill development and real-world project exposure.