**1. What is ReactJS?**

**Answer:**  
ReactJS is an **open-source JavaScript library** developed by Facebook (2011, public release in 2013).

* It focuses only on the **View (UI) layer** of applications in the **MVC (Model-View-Controller)** architecture.
* It allows developers to build **reusable UI components** that manage their own state.
* React is **declarative** → we tell React *what* UI should look like, and React handles the DOM updates efficiently.
* React enables **Single Page Applications (SPA)**, where navigation feels instant because React updates the **view dynamically** without full-page reloads.

 In React, you **describe what the UI should look like**, and React takes care of updating the DOM.

 You don’t write step-by-step DOM manipulation code.

 React internally uses the **Virtual DOM + diffing algorithm** to figure out the minimal changes needed.

2.

**JavaScript (JS)**

**Definition:**  
JavaScript is a **high-level, interpreted programming language** used to create dynamic and interactive content on web pages. It can manipulate the **DOM**, handle events, perform logic, and communicate with servers. JavaScript runs directly in browsers or on servers (Node.js).

**JavaScript (vanilla) is Imperative**

* In plain JavaScript, you must give the **exact steps** to update the DOM.
* The browser only does what you tell it, line by line.

**3.JSX (JavaScript XML)**

**Definition:**  
JSX is a **syntax extension for JavaScript** that allows developers to write **HTML-like code inside JavaScript**. It is primarily used in **React** to describe the UI in a declarative way. JSX is **not understood by browsers directly**; it is **transpiled into regular JavaScript** using tools like Babel before execution.

To embed any JavaScript expression in a piece of code written in JSX we will have to wrap that expression in curly braces {}.

const name = "Learner";  
const element = (  
 <h1>  
 Hello,  
 {name}.Welcome to GeeksforGeeks.  
 </h1>  
);

**How do browsers read JSX?**

In general, browsers are not capable of reading [JSX](https://www.geeksforgeeks.org/reactjs/reactjs-jsx-introduction/) and only can read pure JavaScript. The web browsers read JSX with the help of a transpiler. Transpilers are used to convert JSX into JavaScript. The transpiler used is called [Babel](https://www.geeksforgeeks.org/javascript/what-is-babel/).

**4.**Declarative Language

**Definition:**  
A **declarative language** is a programming style where you **describe *what* you want to achieve**, rather than **how to do it step by step**.

* You focus on **the result**, not the procedure.
* Examples in web development: **HTML, SQL, React (JSX)**

**Example in React (declarative):**

// You declare \*what\* the UI should look like

const element = <h1>Hello, World!</h1>;

Here, React figures out **how to render it** in the DOM — you don’t manually create elements or append them.

**5.Imperative language:**

* In **imperative programming** (like plain JavaScript), if you want to add something to the web page, you must **manually**:
  1. Create an element (document.createElement("h1")).
  2. Add text to it (element.textContent = "Hello")
  3. Insert it into the page (document.body.appendChild(element)).
* In **React’s declarative style (with JSX)**, you just **declare what the UI should look like**:
* const element = <h1>Hello</h1>;

You don’t explain step by step *how to build it*.

* React’s **engine (Virtual DOM + React.createElement)** takes care of:
  1. Creating the actual element,
  2. Adding content,
  3. Deciding where to place it in the DOM,
  4. Updating it later if state changes.

6.Single Page Application:

**Definition**

A **Single Page Application (SPA)** is a type of web application where:

* The browser loads a **single HTML page** initially.
* As the user navigates, only the **content of the page updates dynamically** using JavaScript (via frameworks like React, Angular, Vue).
* There are **no full page reloads**.

**🔹 How it Works**

1. On the first visit → the browser downloads a single index.html, plus CSS + JavaScript.
2. After that → all navigation and content changes happen dynamically using JavaScript and APIs.
3. The app **updates only the part of the page that changes**, instead of reloading the entire page.

**React Router for Navigation**

* To make it behave like a multi-page site, React uses **React Router**.
* It changes the **URL in the browser** but does not reload the page.
* Instead, it **renders the matching component** dynamically

7.MVC:

**Model (Data & Business Logic)**

In React apps, the **Model** is usually:

* **State** → Local component state (useState, useReducer).
* **Global State Management** → Redux, Context API, Zustand, MobX.
* **Backend APIs** → Data fetched from servers (REST APIs, GraphQL).

**. View (UI Layer)**

React itself is the **View layer** in MVC.

* Components describe **what the UI looks like** for given state/props.
* JSX makes writing Views easy and declarative.

// View = UI that displays model data

function TodoList({ todos }) {

return (

<ul>

{todos.map(todo => <li key={todo.id}>{todo.text}</li>)}

</ul>

);

}

**Controller**

* Acts as a **middleman** between Model and View.
* Takes **user input** from the View, updates the Model, and refreshes the View.

📌 Example: When a user clicks “Add to Cart”:

* Controller receives the event → tells the Model to update cart data → tells View to re-render the cart.

**In React, MVC means:**

* **Model** = state & data (from props, APIs, Redux),
* **View** = React components (JSX UI),
* **Controller** = event handlers & functions that update state.

**8. Explain the building blocks of React.**

The five main building blocks of React are

* **Components:** These are reusable blocks of code that return [HTML](https://www.geeksforgeeks.org/html/html-tutorial/).
* **JSX:** It stands for JavaScript and XML and allows you to write HTML in [React](https://www.geeksforgeeks.org/reactjs/reactjs-introduction/).
* **Props and State:** props are like function parameters and State is similar to variables.
* **Context**: Context in React is a way to share data (like theme, user info, or settings) across components without having to pass props manually at every level.

📌 **Why we need it?**  
Normally, data flows **parent → child → grandchild** via props (called **prop drilling**). Context removes this problem by providing a **global-like store** that any component can access directly.

**Virtual DOM:** It is a lightweight copy of the actual DOM which makes DOM manipulation easier.

**9.Explain props and state in React with differences**

Props are used to pass data from one component to another. The state is local data storage that is local to the component only and cannot be passed to other components.

**10.How do you pass data from parent to child using props?**  
By adding attributes in the child component call.

<Child name="Vinu" />

**11. Can props be changed inside a child component?**  
No, props are immutable.

**One-way data flow:**  
React enforces **unidirectional (top-down) data flow**.

* Parent → Child via props.
* If children could directly modify props, data would become unpredictable.

Lifiting State up:

Props themselves are **read-only**, but with the help of an **updater function** (passed down as a prop), the **child can request changes** to the parent’s state.

This is the standard React pattern → called **“lifting state up”**.

function Parent() {

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

return <Child count={count} updateCount={setCount} />;

}

function Child({ count, updateCount }) {

return (

<div>

<h1>{count}</h1>

<button onClick={() => updateCount(count + 1)}>Increment</button>

</div>

);

}

low here:

* count (state) lives in **Parent**.
* Child **receives count (read-only)** and an **updater function** updateCount.
* Child cannot **mutate props directly**, but by calling the updater, it tells the Parent: *“please update your state.”*
* Parent re-renders with new state → passes updated prop down again.

**12.Browsers**: The browser’s rendering engine (e.g., Blink, Gecko) is responsible for taking the updated real DOM and rendering it visually. The browser itself has no understanding of the Virtual DOM concept, only the actual DOM, CSS, and rendering.

* **javaScript Frameworks/Libraries**: The Virtual DOM is created and managed by frameworks or libraries in your JavaScript code. These tools keep a virtual copy of the DOM in memory, calculate the differences ("diffing") between current and new UI states, and decide which real DOM parts should be updated.

**Real DOM**

* Managed by the **browser engine** (like Chrome’s Blink, Firefox’s Gecko).
* It’s the **actual UI tree** the browser uses to draw pixels on screen.
* Stored in browser’s memory, but tightly integrated with **layout, CSS, rendering, and painting engine**.
* Every update to Real DOM triggers heavy operations → style recalculation, layout reflow, repaint.

**🔹 Virtual DOM**

* Managed by **React (JavaScript engine, not browser engine)**.
* It’s just a **plain JavaScript object** — like a JSON representation of your UI.
* {
* type: "h1",
* props: { id: "count", children: "0" }
* }
* Exists **only in JS runtime memory**, not connected to the browser’s rendering pipeline.
* Super lightweight → changes are cheap.

**🔹 How they connect**

1. React renders JSX → creates a **Virtual DOM tree in JS memory**.
2. React translates Virtual DOM → **Real DOM nodes in browser** (initial render).
3. On updates:
   * React builds a **new Virtual DOM tree** (in JS memory).
   * Compares with old V-DOM (also in JS memory).
   * Applies **minimal changes** to the Real DOM (browser memory).

✅ So the correct way to phrase it:

* **Both Real DOM and Virtual DOM live in memory.**
* **Difference:** Real DOM is the browser’s memory structure (tied to rendering engine), while Virtual DOM is React’s memory structure (just JS objects).
* **Interview-ready one-liner:**  
  Virtual DOM is a lightweight JS copy of the Real DOM that React manages in JS memory, while the Real DOM is the browser-managed structure used to render the UI on screen.

**13.**

**Diffing = comparing the old Virtual DOM with the new Virtual DOM**

* When your component’s state/props change, React builds a **new Virtual DOM tree**.
* React then **compares** this new tree with the previous one.
* This process is called **diffing**.
* Goal: figure out exactly **what changed** (e.g., text node, attribute, or child element).

⚡ React uses an **optimized diffing algorithm**:

1. If an element type changes (<div> → <p>), React destroys the old and creates a new node.
2. If only attributes change (e.g., className, id), React only updates those.
3. If children in a list change, React uses **keys** to efficiently match items.

**14.Reconciliation in React**

**Reconciliation = applying the minimal changes found during diffing to the Real DOM.**

* After diffing, React knows *where* the differences are.
* Then React updates the Real DOM **only at those places** (not the whole tree).
* This process of updating the Real DOM with the diff result is called **reconciliation**.

function App() {

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

return <h1>{count}</h1>;

}

1. Initial render → Virtual DOM:  
   { type: "h1", props: { children: "0" } }  
   → Real DOM: <h1>0</h1>
2. After setCount(1) → New Virtual DOM:  
   { type: "h1", props: { children: "1" } }
3. **Diffing:** React compares → Only text changed ("0" → "1").
4. **Reconciliation:** React updates only that text node in the Real DOM.

✅ So instead of repainting the whole <h1> or whole page, React only updates "0" → "1".

* **Diffing:** Comparing the old and new Virtual DOM to find what changed.
* **Reconciliation:** Applying those minimal changes to the Real DOM.

15.

**Differentiate between Real DOM and virtual DOM?**

| **Real DOM** | **Virtual DOM** |
| --- | --- |
| The actual DOM, a tree-like structure representing the UI elements. | A lightweight copy of the Real DOM used to optimize updates. |
| Slower as it requires direct updates to the actual DOM. | Faster because it minimizes direct manipulation of the Real DOM. |
| Directly manipulates the Real DOM, causing re-rendering of the entire UI. | Updates are made to the Virtual DOM first, then changes are batched and only the necessary changes are reflected in the Real DOM. |
| Entire UI might need to be re-rendered when changes occur. | Only the necessary components are re-rendered, reducing unnecessary re-renders. |
| Less efficient due to repeated direct updates to the Real DOM. | More efficient by minimizing direct DOM manipulation and batch updates. |

**16.What are components and their type in React?**

A [Component](https://www.geeksforgeeks.org/reactjs/reactjs-components/) is one of the core building blocks of React. In other words, we can say that every application you will develop in React will be made up of pieces called components. Components make the task of building UIs much easier.

**In React, we mainly have two types of components:**

* **Functional Components:**Functional components are simply JavaScript functions. Initially, they were limited in terms of features like state and lifecycle methods. However, with the introduction of Hooks, functional components can now use state, manage side effects, and access other features that were once exclusive to class components.
* **Class Components:** Class components are more complex than functional components. They are able to manage state, handle lifecycle methods, and can also interact with other components. Class components can pass data between each other via props, similar to functional components.

**17.What is a Side Effect then?**

**side effect** is when your component does something **other than showing UI**.

👉 Like:

* Fetching data from an API.
* Saving to localStorage.
* Subscribing to a WebSocket.
* Setting a timer.
* Changing the DOM directly.

These are “side effects” because they affect the **outside world**, not just the return value.

These are “extra effects” → not just returning UI.  
That’s why they are called **side effects**.

A React component’s job is simple:  
👉 Take **props + state** → return **UI (JSX)**. But side effect is not that

18.

**useEffect — the basics**

useEffect(() => {

// effect: runs after render (default)

return () => {

// cleanup: runs before next effect or on unmount

};

}, [dependencies]);

* No deps (useEffect(() => {...})) → runs **after every render**.
* Empty deps (useEffect(() => {...}, [])) → runs **once after mount** (like componentDidMount).
* With deps ([a, b]) → runs **when any dependency changes**.

**Do not** put side effects directly in component body — the body runs every render and would cause repeated side-effects.

19.

React Lifecycle:

**Moutning:**

Mounting is the phase when a React component is created and inserted into the DOM, and code that should run after first render is placed inside useEffect(..., []).

**componentWillMount (Class Concept)**

* **What it was:** Called **before the component is rendered** on the DOM.
* **Status:** Deprecated in React 16+ → **don’t use it anymore**.
* **Why:** Code inside it could cause bugs because the component isn’t yet in the DOM.

**Function Component Equivalent:**

* There’s **no exact equivalent**.
* Anything that needs to run **before first render** is usually done **inside initialization** or **outside of useEffect**, like setting initial state.

const [count] = useState(0); // initial value before render

**componentDidMount**

* **What it is:** Called **after the component is rendered** on the DOM.
* **Use:** Run side effects like:
  + API calls
  + Setting timers
  + Subscribing to events

**Updating Phase (Function Components)**

**Definition:**  
Updating happens when a component **re-renders due to a change in state or props**.

**Key points:**

* Happens **every time state or props change**.
* Use useEffect with **dependencies** to run code on updates.

import { useState, useEffect } from "react";

function Counter({ step }) {

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

// Runs only when 'count' changes → updating

useEffect(() => {

console.log("Count updated:", count);

}, [count]);

return (

<div>

<h1>{count}</h1>

<button onClick={() => setCount(count + step)}>Increment</button>

</div>

);

}

**Explanation:**

* Every time setCount updates count, component re-renders.
* useEffect sees count changed → runs the effect → logs updated count.
* This is **function component equivalent of componentDidUpdate** in class components.

**2️⃣ Unmounting Phase (Function Components)**

**Definition:**  
Unmounting happens when a component is **removed from the DOM**.

**Use:** Clean up side effects to avoid memory leaks.

* Timers (setInterval, setTimeout)
* Event listeners (window.resize, keyboard events)
* API subscriptions or WebSockets

**Definition (Class Component):**  
componentWillUnmount is a lifecycle method called **just before a component is removed from the DOM**.

**20.What is useState?**

**Definition:**  
useState is a **React Hook** that allows **function components to have state**.

* In class components, state was only available via this.state and this.setState.
* In function components, useState gives you **state variables** and a **setter function** to update them.

**Syntax:**

const [state, setState] = useState(initialValue);

* state → current value of the state.
* setState → function to update the state.
* initialValue → the value state starts with

**21.Can you store objects or arrays in useState?**

**Yes.**  
useState can store **any type of JavaScript value**:

* primitives (numbers, strings, booleans)
* arrays
* objects
* even functions

When updating **arrays or objects**, you **must not mutate them directly**.

* **Why?**  
  React compares state **by reference**.
* If you mutate the object/array directly, React may **not detect changes**, and the component **won’t re-render** properly.

const [items, setItems] = useState(["Apple", "Banana"]);

// Add a new item

const addItem = () => {

setItems(prevItems => [...prevItems, "Orange"]); // ✅ use new array

};

 ...prevItems → spreads existing array.

 [...prevItems, "Orange"] → creates a **new array reference**.

 React sees the **new reference** → re-renders component.

**Bad example (mutating):**

items.push("Orange"); // ❌ mutates original array

setItems(items); // React may not re-render

**22.When to use useRef here:**

* You **don’t need to update the UI** on every input change.
* Only need the value **at the time of submit**.

**23.What is useMemo?**

**Definition:**  
useMemo is a React Hook that **memorizes the result of a function** and only **recomputes it when its dependencies change**.

* Useful to **optimize expensive calculations**.
* Prevents **unnecessary recalculations** on every render.

const doubled = useMemo(() => number \* 2, [number]); // recalculates only if 'number' changes

**Explanation:** doubled is memoized and won’t recalculate unless number changes.

**24.What is useContext?**

**Definition:**  
useContext is a React Hook that lets a component **access values from a Context** without passing them as props through every level (avoids prop drilling).

* Works with **React Context** (React.createContext).
* Returns the **current context value**.
*  ThemeContext is created with default value "light".
*  App provides "dark" as the current value.
*  Child uses useContext(ThemeContext) to **read the value directly**.

 Avoids **prop drilling**: no need to pass props through multiple levels.

 useContext **reads** the current context value;

useContext lets function components access values from a Context directly, avoiding prop drilling."

**25.What is React Fragment?**

**Definition:**  
A **React Fragment** is a way to group **multiple elements** without adding an extra **DOM node** like a <div>.

* Syntax:

<React.Fragment> ... </React.Fragment>

or shorthand:

<> ... </>

* Helps **avoid unnecessary wrappers** in the DOM.

**If Not Used**

* React **requires a single parent element** to return from a component.
* Without a wrapper (div or Fragment), you get this **error**:

JSX expressions must have one parent element

* People often use <div> to fix it, but it adds **extra nodes** in the DOM, which can break CSS/layouts or cause unnecessary nesting

26.React developer tool:

React Developer Tools is a browser extension that allows inspecting the React component tree, viewing props, state, and hooks, and debugging React applications efficiently."

**What is conditional rendering in React?**

[Conditional rendering](https://www.geeksforgeeks.org/reactjs/reactjs-conditional-rendering/)in React is used when you want to render different UI elements based on certain conditions. For example, rendering a login button if a user is not logged in or rendering a logout button when the user is logged in.

**const** isLoggedIn = **true**;

**return** (

<div>

{isLoggedIn ?<button>Logout</button> : <button>Login</button>}

</div>

);

**27.What are Controlled Components?**

**Simple Definition:**  
A **controlled component** is a form element (like <input>, <textarea>, <select>) **whose value is controlled by React state**.

The form element **always reflects the state**.

import { useState } from "react";

function Controlled() {

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

return (

<input

type="text"

value={name} // input value comes from state

onChange={e => setName(e.target.value)} // updates state

/>

);

}

**28.Uncontrolled Component**

**Definition:**

* A component where **the DOM itself controls the input value**.
* React does **not track every change**. You read the value **only when needed**, usually via ref.

import { useRef } from "react";

function Uncontrolled() {

const inputRef = useRef();

const handleSubmit = () => {

alert("Input value: " + inputRef.current.value); // read value directly from DOM

};

return (

<div>

<input type="text" ref={inputRef} />

<button onClick={handleSubmit}>Submit</button>

</div>

);

}

"Controlled components are managed by React state, while uncontrolled components are managed by the DOM itself and read using refs."

**29.What is .then()?**

**Definition:**  
.then() is a **method of a Promise** that lets you **run code after an asynchronous operation completes successfully**.

* It receives a **callback function** with the **result of the promise**.
* Runs **only when the promise is fulfilled** (not rejected).

In Axios:

A **JavaScript library** for making HTTP requests, with **automatic JSON parsing** and **simpler syntax**.

axios.get("https://api.example.com/data")

.then(res => console.log(res.data)) // runs after request succeeds

.catch(err => console.error(err));

**Explanation:**

* axios.get() returns a **Promise**.
* .then(res => ...) runs **after Axios gets the response**.
* res.data contains the JSON data from the server.

**30.Fetch**

**Definition:**  
A **built-in browser API** for making HTTP requests, but you must **manually parse JSON** and **handle errors**.

fetch("https://api.example.com/data")

.then(res => res.json()) // runs after server responds

.then(data => console.log(data)) // runs after JSON is parsed

.catch(err => console.error(err));

1. fetch() returns a **Promise**.
2. First .then(res => res.json()) → parse response to JSON.
3. Second .then(data => ...) → now you have the actual data.

**31.What is CORS?**

**Definition:**  
**CORS (Cross-Origin Resource Sharing)** is a **security feature in browsers** that **blocks web pages from making requests to a different domain** than the one that served the page, unless the server explicitly allows it.

* Browsers restrict requests from http://localhost:3000 to another domain like http://api.example.com unless the server **permits cross-origin requests**.
* It’s a **browser security feature**, not React-specific.

**32.What is React Material-UI?**

**Definition:**  
**React Material-UI** (now called **MUI**) is a **popular React component library** that provides **pre-built, ready-to-use UI components** based on **Google’s Material Design**.

* It allows developers to **build beautiful, responsive, and consistent UIs quickly**.
* Components are **customizable** and follow **Material Design guidelines**.

**🔹 Key Features**

1. **Pre-built Components:** Buttons, TextFields, Cards, Dialogs, AppBar, etc.
2. **Customizable Themes:** Change colors, typography, spacing, and other styles globally.
3. **Responsive Design:** Components adapt to different screen sizes automatically.
4. **Accessibility:** Built-in ARIA support and keyboard navigation.
5. **Easy Integration:** Works seamlessly in React projects.

33.What is strict mode?

**React Strict Mode** is a development tool that **helps detect potential problems** in an app by running checks **twice** without affecting the UI.

import React from "react";

import ReactDOM from "react-dom/client";

import App from "./App";

ReactDOM.createRoot(document.getElementById("root")).render(

<React.StrictMode>

<App />

</React.StrictMode>

);

**Explanation:**

* <React.StrictMode> wraps your app.
* React **runs certain checks twice** in development to detect potential problems.
* No effect on UI in the browser.

**34.What is Lazy Loading in React?**

[Lazy Loading](https://www.geeksforgeeks.org/reactjs/lazy-loading-in-react-and-how-to-implement-it/) in React is a technique used to load components only when they are needed, instead of loading everything at once when the app starts. This helps improve the performance of the app by reducing the initial loading time.

In React, React.lazy() is used to implement lazy loading for components, which allows you to split your code into smaller bundles and load them only when required.

**35.What is a Higher-Order Component?**

**Definition (Simple):**  
A **Higher-Order Component (HOC)** is a **function that takes a component as input and returns a new enhanced component**.

* It’s a **pattern** to reuse component logic.
* Think of it as **wrapping a component to add extra features** without modifying the original component.

**36.Difference between map and forEach**

**Question:** When to use map vs forEach?

**Answer:**

* map → **returns a new array** after applying a function to each element.
* forEach → **does not return anything**, just iterates over the array.

const arr = [1, 2, 3];

const doubled = arr.map(x => x \* 2); // [2, 4, 6]

arr.forEach(x => console.log(x)); // print 1 2 3

**37.Filter an array**

**Question:** How to get only even numbers from an array?

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

const even = arr.filter(x => x % 2 === 0); // [2, 4]

* filter → **returns a new array** with elements that satisfy the condition.

**38.Reduce an array**

**Question:** How to sum all numbers in an array?

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

const sum = arr.reduce((acc, curr) => acc + curr, 0); // 10

* reduce → **reduces array to a single value** using a callback function.

**39.Find an element**

**Question:** How to find the first number greater than 3?

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

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

* find → returns **first element** that satisfies the condition.

**40.Chaining methods**

**Question:** Can you chain map, filter, and reduce?

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

const result = arr

.filter(x => x % 2 === 0) // [2, 4]

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

.reduce((acc, curr) => acc + curr, 0); // 12

**41.What is Event Delegation?**

**Definition:**  
Event delegation is a **technique where a single event listener is attached to a parent element** to handle events on its **child elements**, instead of adding separate listeners to each child.

* The **event bubbles up** from the target element to the parent.
* React uses **Synthetic Events** to implement this efficiently.

**Event Bubbling in React**

* **Bubbling**: Events start at the **target element** and move up to **parent elements**.
* React wraps **all native events in SyntheticEvent**, so bubbling works consistently across browsers.

**42.What is Hydration?**

**Definition:**  
Hydration is the process where **React takes HTML that was pre-rendered on the server (SSR)** and **“attaches” React’s event listeners** to make it fully interactive on the client side.

* The page **looks ready** because the HTML is already there.
* React **hydrates** it by adding event handling and state management.

**2️⃣ Why Hydration is Needed**

When you use **Server-Side Rendering (SSR)**:

1. The server sends **HTML content** to the browser → page is visible immediately.
2. But that HTML **doesn’t have React events** attached.
3. Hydration is the **process of linking React to that HTML** so it becomes a **fully interactive React app**.

**Client-side hydration:**

// Client

ReactDOM.hydrate(

<App />,

document.getElementById("root")

);

* hydrate tells React: “The HTML is already here; just attach event listeners and make it interactive.”
* Without hydration, buttons, forms, and other interactive elements **won’t work**.

**43.Client-Side vs Server-Side Rendering**

| **Feature** | **Client-Side Rendering (CSR)** | **Server-Side Rendering (SSR)** |
| --- | --- | --- |
| Where HTML is generated | Browser (client) | Server |
| Initial load speed | Slower | Faster |
| SEO | Poor | Better |
| React hydration | Not needed | Required to attach event listeners |