**Client-Side Rendering (CSR)**

**Analogy**: Amazon sends you **all the parts** of the table in a box with a manual. You have to **assemble** it yourself at home.

| **Aspect** | **CSR (Client-Side Rendering)** |
| --- | --- |
| What Amazon sends | Just the parts and an instruction manual |
| Your task (Client) | You assemble the table from parts |
| Initial experience | Slow — takes time to build and see the full table |
| After first build | Fast interactions — you don’t need to reassemble again |
| Example | React app loading a bare page and fetching data via API |
|  |  |

**Server-Side Rendering (SSR)**

**Analogy**: Amazon sends you the **fully assembled** table. You just open the box and it’s ready to use.

| **Aspect** | | **SSR (Server-Side Rendering)** | |
| --- | --- | --- | --- |
| What Amazon sends | | Fully assembled table | |
| Your task (Client) | | Just unpack it and start using | |
| Initial experience | | Fast — the table is ready right away | |
| After first load | | Slower if you need to reorder — every table comes reassembled | |
| Example | | HTML page generated on the server (e.g., Next.js SSR) | |
| **Use Case** | **CSR** | | **SSR** | |
| Product Table Listing | JS fetches product data and assembles table on browser | | Server sends complete HTML table ready to display | |
| Sorting/Filtering | Fast (no reload, just update data in browser) | | May need full page reload (unless hybrid SSR+JS used) | |

**Server-side rendering (SSR)** is the process of rendering web pages on the server and sending the fully-rendered HTML to the client. In this approach, the server generates the HTML, including any dynamic data, and sends it to the client as a complete page. The client then displays the page without any further processing.

**Client-side rendering (CSR)** is the process of rendering web pages on the client using JavaScript. In this approach, the server sends the initial HTML file, but the client then uses JavaScript to dynamically update the page as needed. This allows for more interactive and responsive web pages, as the client can update specific parts of the page without needing to reload the entire page.

One example of a popular CSR framework is React. With React, you can write JavaScript code that updates the DOM as needed, providing a more interactive and dynamic web application.

**What is a Single Page Application (SPA)?**

A **Single Page Application** is a **web app or website** that:

* Loads **a single HTML page** initially.
* Dynamically updates the content **without refreshing** the whole page.
* Uses **JavaScript (usually frameworks like React, Angular, or Vue)** to fetch and display data.

**Component-based architecture** means breaking the UI into small, reusable parts called **components**.  
Each component handles its own structure, style, and logic.  
In Instagram, components like <Post />, <Comment />, and <LikeButton /> combine to form the user feed.  
This makes the app easier to build, maintain, and scale.

**Syntax to create :**

**Case 1: Create project in a new folder**

npm create vite@latest my-app

This creates a new folder called my-app and puts your Vite project **inside it**.

**✅ Case 2: Create project in current folder**

npm create vite@latest .

This initializes the Vite project **inside the current folder** (⚠️ make sure the folder is empty or you'll get warnings).

To run the project:

1.npm install is used to **download and install** all the dependencies listed in a project's package.json file.

2.npm run dev

**In a Vite + React app:**

* There's only **one HTML file**: index.html
* Inside it, we have an empty <div id="root"></div> in the body
* This is the **placeholder** where React will render the UI

In main.jsx,

createRoot(document.getElementById('root')).render(

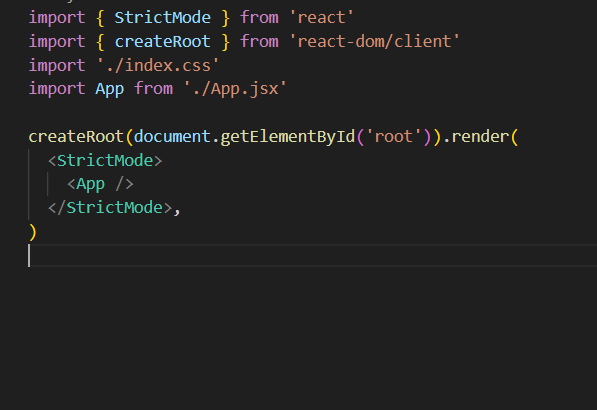
<StrictMode>

<App />

</StrictMode>

)

* Finds the empty div with id root
* Injects the entire React component tree (<App />) into it
* So, React controls everything inside that one div dynamically.

A screen shot of a computer program

AI-generated content may be incorrect.

**TYPES OF EXPORT:**

**Named Export**

* You can export **multiple items**
* You **must use the same name** when importing

// utils.js

export const add = (a, b) => a + b;

export const sub = (a, b) => a - b;

// main.js

import { add, sub } from './utils.js';

**. Default Export**

* You can export **only one item** as default from a file
* You can import it with **any name**

// math.js

const multiply = (a, b) => a \* b;

export default multiply;

// main.js

import mul from './math.js'; // Can name it anything

3.Why Components Start with Capital Letters in React:

function H1() {

return <h1>This is a component!</h1>;

} function App() {

return (

<div>

<H1 /> {/\* ✅ Custom Component \*/}

<h1>Heading</h1> {/\* ✅ HTML Element \*/}

</div>);}

React uses this naming rule to **differentiate between your own components and built-in HTML tags**.

**Why can’t we return multiple elements in JSX?**

 React builds a **tree** of elements (called the Virtual DOM).

 A tree must have **only one root** — like a folder must start from one top-level directory.

 If we return multiple elements **without wrapping**, React doesn’t know which one is the "main parent" to attach to the page.

**❌ Not allowed:**

return (

<h1>Hello</h1>

<p>World</p>

);

* ❌ React says: "I see two things. I need just **one** root element."

Allowed (wrapped in one tag):

return (

<div>

<h1>Hello</h1>

<p>World</p>

</div>

);

use **Fragments** (don’t add extra HTML):

jsx

CopyEdit

return (

<>

<h1>Hello</h1>

<p>World</p>

</>

);

**4.Using JavaScript inside JSX**

Use **curly braces {}** to write JS expressions inside JSX.

const name = "React";

return (

<h1>Hello, {name}!</h1>

);

**What are Props?**

**Props** = **"Properties"**  
They are **parameters passed to components** — just like function arguments.

Think of a component as a **function**, and **props** are the **inputs** to that function.

**Why use props?**

* To **pass data** from a **parent component** to a **child component**
* Props make components **dynamic** and **reusable**

How props work:

**1. Pass props from parent:**

App name="John" age={25} />

2. **Receive props in the component:**

function App(props) {

// U can do like const{name,age}=props:

return (

<div>

<h1>Hello, {props.name}</h1>

<p>Age: {props.age}</p>

</div>

);

}

3**.Instead we can destructure it**:

function App({ name, age }) {

// u can use const{name,age}=props aslo

return (

<div>

<h1>Hello, {name}</h1>

<p>Age: {age}</p>

<div> );} Props passed from parent should match child with same name for exampke if name={john} then in child it should be props.name not props.anyvariable

Why does React render twice?

In development mode (not in production), React wraps your components in:

<React.StrictMode>

<App />

</React.StrictMode>

 React renders your component once.

 Then **immediately renders it again** (but not visually).

 This helps catch issues **before** they cause bugs in real usage.

This double render only happens in **development**, not in the final **production build** — so users won’t experience it.

**Why can't we directly render an object in React?**

In JSX, React can only render **strings, numbers, or valid JSX elements**.

So this:

jsx

CopyEdit

const person = { name: "John" };

return <div>{person}</div>; // ❌ Error!

❌ You’ll get: Objects are not valid as a React child

Correct return <div>{person.name}</div>;

Immutabily in props:

**Props are read-only** → You **cannot change** them inside the child component.

Because props come **from the parent**. The child should only **use** them, not **modify** them.

Example:

function App(props) {

props.name = "Changed"; // ❌ This will give a warning

return <h1>{props.name}</h1>;

}

If array is directly rendered:

const arr = ['apple', 'banana', 'orange'];

return <h1>{arr}</h1>; // Output: applebananaorange

it is displayed without commas

Pass Object as Props:

Parent Child:

|  |  |
| --- | --- |
| import Child from './Child';  function App() {  const user = {  name: 'Alice',  age: 22,  city: 'Chennai'  };  return (  <div>  <Child userInfo={user} />  </div>  );  }  export default App; | function Child(props) {  return (  <div>  <h2>Name: {props.userInfo.name}</h2>  <p>Age: {props.userInfo.age}</p>  <p>City: {props.userInfo.city}</p>  </div>  );  }  export default Child;  Name: Alice  Age: 22  City: Chennai  **Explanation:**   * user is an **object** with properties like name, age, and city. * It's passed to the child as a prop: userInfo={user} * In the child, we access it using props.userInfo.name, props.userInfo.age, etc. |

The above method u destruture with props variable but we can do more Easily

|  |  |
| --- | --- |
| import Child from './Child';  function App() {  const user = {  name: 'Alice',  age: 22,  city: 'Chennai'  };  return (  <div>  <Child user={user} />  </div>  );  }  export default App; | function Child({ user }) {  const { name, age, city } = user;  // return(<><h1> user.name</h1><>);  return (  <div>  <h2>Name: {name}</h2>  <p>Age: {age}</p>  <p>City: {city}</p>  </div>  );  }export default Child;  Instead of props.user, we directly extract user from props:  function Child({ user }) { ... }   * Then we destructure name, age, and city from user. |

How to render array or list:

function App() {

const fruits = ['apple', 'banana', 'orange']

return (

<div>

<h2>Fruit List:</h2>

<ul>

{fruits.map((fruit, index) => (

<li key={index}>{fruit}</li>

))}

</ul>

</div>

);

export default App;

fruits.map(fruit=>console.log(fruit)} In this , fruits.map() will map each single item to fruit variable and with the help of arrow function we render

How to render Array of object:

|  |  |
| --- | --- |
| function App() {  const users = [  { id: 1, name: 'Alice', age: 22 },  { id: 2, name: 'Bob', age: 25 },  { id: 3, name: 'Charlie', age: 28 }  ];  return (  <div>  <h2>User List:</h2>  <ul>  {users.map(user => (  <li key={user.id}>  {user.name} - Age: {user.age}  </li>  ))}  </ul>  </div>  );  }  export default App;  key={user.id} to give each element a unique key (React needs this). | function App() {    const users = [      {  name: 'Alice', age: 22 },      {  name: 'Bob', age: 25 },      {  name: 'Charlie', age: 28 }    ];    return (      <div>        <h2>User List:</h2>        <ul>          {users.map(user => (            <li >              {user.name} - Age: {user.age}            </li>          ))}        </ul>      </div>    );  }  export default App;  When a list updates (add, remove, reorder), React uses the key to **know which item changed** and avoid re-rendering the entire list.  Warning: Each child in a list should have a unique "key" prop. |