Server-Side rendering:

The **server** generates the **complete HTML** for a webpage
That HTML is sent to the **browser**
The browser displays the page immediately

Instead of sending an empty HTML + JavaScript bundle (like in Single Page Application React apps), the server directly returns ready-made HTML.

Diagram:
Client → Server → Server fetches data → Server builds HTML → Sends HTML → Browser displays it

Why SSR Exists?

SSR solves two main problems:

- SEO needs: Search engines require **fully rendered HTML** to index pages properly.
- Faster First Contentful Paint (FCP): Because HTML is ready, the browser shows content immediately.
- Better for low-end devices: Rendering happens on server → client CPU does less work.
- Faster first-page load (Better SEO + better user experience): Search engines can read fully formed HTML → huge benefit for SEO.
- Works even if the user has slow devices: HTML is rendered on the server → browser does less work.
- Better for content-heavy pages: Blogs, news websites, dashboards, ecommerce categories, etc.

When NOT to use SSR?

SSR does **extra computation** on the server → may increase server cost.
Not good for:

- Highly interactive dashboards
- Real-time apps
- Pure single-page apps (React SPA with client-side routing)

Why SSR is not good for highly interactive dashboards?
Dashboards update many times (charts, metrics, notifications).

SSR problem:
SSR generates HTML **on the server** for every request.
But dashboards need **frequent updates on the client**, not a full-page render.

a. **Why SSR fails here:**

- Too many server requests → **expensive**
- UI updates **every second** → SSR cannot re-render server-side continuously
- Server will get overloaded
- Lots of network traffic (full HTML repeatedly)

Dashboards need CSR (Client-Side Rendering) because:

- Only required parts update using JavaScript
- Faster real-time updates
- No need to reload HTML from server

b. Why SSR is not good for real-time apps?

Examples:

- Chat applications
- Live notifications
- Stock prices
- Live dashboards
- Multiplayer games

Real-time apps use:

- WebSockets
- EventSource
- Long polling
- Live data streams

Why SSR fails here:

SSR sends **static HTML** once.
But real-time apps need **constant UI updates**
You cannot ask server to re-render the whole HTML again every second.
SSR is **not built for continuous updates**
Server re-renders full HTML page → sends full HTML to browser → expensive + slow
This kills performance.

CSR is better because:

- JavaScript updates only the required components
- Much less server work
- Much better performance

c. Why SSR is not good for pure SPAs (React SPA): Single page application

The browser renders everything using JavaScript.
Examples:

- Instagram
- Tello
- Gmail
- Todo App
- Slack

SPAs are designed to:

- Load once
- Navigate without page reload
- Update UI dynamically

Why SSR does not help SPAs:

- SPA pages cannot rely on server HTML because SPA uses **client-side routing**
- SPA loads a huge JS bundle anyway → SSR does not reduce JS size
- SSR becomes unnecessary overhead

How SSR Works? (Flow):
Client (Browser) → HTTP Request → Node.js Server
Node.js Server → Fetch data → Render HTML → Send to browser
Browser → Displays complete HTML page immediately

Example:

SSR in Node.js (Using Express + EJS Example)

1. npm install express ejs
2. folder structure:

```
project
 ├── views
 │    └── home.ejs
 └── app.js
```

3. **app.js – Server Code**

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

app.set("view engine", "ejs");

app.get("/", (req, res) => {
  const user = {
    name: "Bibek",
    age: 23
  };

  res.render("home", { user });
});

app.listen(3000, () => {
  console.log("SSR server running on http://localhost:3000");
});
```

4. **home.ejs – Template File**

```
<!DOCTYPE html>
<html>
<head>
  <title>SSR Example</title>
</head>
<body>
  <h1>Hello <%= user.name %>!</h1>
  <p>Your age is <%= user.age %> years.</p>
</body>
</html>
```

5. **Result:**
   Hello Bibek!
   Your age is 23 years.

React SSR using Next.js (Modern way):

Next.js internally uses **Node.js + React Server Components + ReactDOMServer**.

Performance Impacts of SSR:

Pros:

- Better SEO
- Fast first paint
- Good for slow devices

Cons:

- More server CPU usage
- Higher hosting cost
- More network load
- Slower when too many dynamic queries

Caching in SSR:

To reduce load:

- Page-level caching (static HTML): CDN (Cloudflare, Akamai, Netlify)
- Data-level caching: Redis, Memory cache, LRU cache
- Fragment caching: Cache only headers/parts of page.
- Revalidation (Next.js)

Streaming SSR:

Instead of generating 100% HTML and sending it at once, the server streams HTML chunks.

Benefits:

- Faster: Time To First Byte: The time it takes for the **browser to receive the very first byte of data** from the server after making a request.
- Browser shows content earlier

Used by: React 18, Next JS 13/14, Node stream

Security Considerations in SSR:

- Escape user input to avoid HTML injection
- Avoid direct HTML injection
- Use HTTP-only cookies for auth
- Use CSRF tokens

React SSR:

- Uses Virtual DOM
- Requires hydration
- Supports streaming
- More CPU usage but powerful

Hydration means:

Hydration happens **after SSR**

When a page is server-rendered:

- The **server** sends complete HTML to the browser
- But that HTML is NOT interactive (buttons do not work yet)
- The **browser downloads JavaScript**
- JavaScript "activates" that HTML by attaching event listeners

What is Streaming?

In **Streaming SSR**: Server sends HTML in small chunks **as soon as each part is ready.** Hence Browser starts showing content sooner

Advantages of SSR:

- SEO-friendly
- Better performance on first load
- Can run without JavaScript
- Easier social media sharing (OpenGraph tags available in HTML)
- Good for content websites

Disadvantages of SSR:

- Slower response time (server must compute HTML)
- Expensive server resources
- Complexity (hydration, caching, streaming, routing)
- Harder to reuse components between server & client without hydration

Client-side Rendering:

The **browser (client)** is responsible for rendering the UI
The **server sends only a basic HTML + JavaScript bundle**
JavaScript **builds and updates the entire UI** on the client side
The page **does not reload** when navigating

CSR is the rendering model used by: React (SPA), Vue, Angular

How CSR works:

1. Browser requests page:
2. Server sends:
    - An empty or minimal HTML file
    - A JavaScript bundle (app.js)
3. Browser downloads JavaScript
   JavaScript:
    - Builds UI
    - Fetches data from APIs
    - Renders components
    - Handles navigation and interactions
4. All future page updates happen **inside the browser** — no full reload.

Detailed internal flow of CSR:

Step1: Browser loads HTML
HTML is mostly empty:

Step2: Browser downloads JS bundle
Contains:

- Components
- Routing logic
- State management
- UI library (React)

Step3: JavaScript runs
JS framework (React/Vue/Angular):

- Creates virtual DOM
- Inserts HTML into #root
- Fetches data from API
- Displays the page

Step4: Client-side routing
When clicking links:

- No full page reloads
- URL updates using history.pushState()
- JS updates only the relevant component

Feels like a mobile App

When CSR is perfect:

- Dashboards
- Admin panels
- Chat apps
- Social networks
- Email clients
- E-commerce admin
- Highly dynamic UIs
- Apps with many components updating frequently

When CSR is bad:
CSR has major disadvantages:

1. Slow First Load:
   CSR needs to download and execute a LOT of JavaScript before showing content.
   User sees:
   - Blank screen
   - Loading spinner

   Because server gave an empty HTML.
2. Very Poor SEO:
   Search engines struggle with JavaScript-rendered pages.
   CSR sites may not index content properly.
3. High JS bundle size makes UX slow:
   More JavaScript → more parsing → more memory usage.