

Server Side Rendering (SSR) is a technique where the HTML for a web page is generated on the server and sent to the browser, instead of relying on the browser to build the page from JavaScript after it loads.

SSR in Simple Terms

- When a user requests a page, the server translates React components into an HTML string and sends it to the browser immediately.
- The browser displays this HTML before downloading and running any JavaScript bundles, so visitors see meaningful content right away.
- After initial HTML is shown, JavaScript loads and attaches (hydrates) React, making the page interactive.

Benefits

- Faster initial load: Users see content sooner, even if their internet is slow.
- Better SEO: Search engines can easily read page content since it's in HTML.
- Improved accessibility: Screen readers don't have to wait for JavaScript to parse the content.

Hydration

After SSR, React “hydrates” the page—attaching event handlers so the page acts like a full React app. This bridges the gap between server-rendered HTML and rich client interactions.

Server-side rendering is especially useful for performance and SEO, and React Router supports it using features like `StaticRouter` and route loaders

React Router v7 brings server side rendering (SSR) features very similar to Remix—since they've effectively merged, you get full-stack capabilities, file-based routing, loaders, actions, middleware, and more, all in React Router itself. Here's how SSR works in React Router v7 and how it compares to Remix:

How SSR Works in React Router v7

- React Router v7 lets you render your app's HTML on the server, just like Remix. [logrocket+2](#)

- The framework mode in v7 enables SSR, data fetching with loaders, mutations with actions, and nested routing out-of-the-box—no need for extra libraries.[remix+1](#)
- You define route modules (e.g., `routes/product.tsx`) where you can export loader functions for data fetching on the server, as well as components to render UI.[probirsarkar+1](#)
- Configuration is straightforward; you set `ssr: true` in your React Router config to enable SSR.[javascript.plainenglish](#)

Example: SSR Route Module

tsx

```
// routes/product.tsx

export async function loader({ params }) {
  // Runs on the server during SSR

  const res = await
fetch(`https://api.example.com/product/${params.id}`);

  const product = await res.json();

  return product;
}

export default function Product({ loaderData }) {
  return <h2>{loaderData.name}</h2>;
}
```

- When someone visits `/product/123`, the server runs the loader above, fetches the product, and renders the Product component to HTML before sending it to the browser.[probirsarkar](#)

Key Features from Remix Included

- **Nested routes:** Build deeply nested layouts, fetch data at any level, and get optimised HTML/SEO output.[optis+1](#)
- **Loaders and actions:** Fetch and mutate data on the server for each route, just like Remix.[remix+1](#)
- **Middleware:** Use route-level middleware for things like authentication, both server and client side.[reactrouter+1](#)
- **File-based routing (optional):** You can keep using programmatic route configs or use filesystem-based routes for convention over configuration.[javascript.plainenglish](#)

Difference With Remix

| Feature | React Router v7 | Remix |
|-----------------------|---|---|
| SSR setup | Manual setup required, flexible | Built-in, less to configure |
| Data loading/mutation | Loaders, actions (optional to use them) | Loaders, actions (required for consistency) |
| Routing | File-based routing, but optional | Always file-based routing |
| Streaming SSR | Supported, but some config needed | Built-in |
| SEO & performance | Must configure caching, SEO | SEO and caching optimised out-of-the-box |

- React Router v7 is flexible and lets you adjust as needed. Remix is more opinionated and “just works” for typical SSR/web app needs.[optis+1](#)

React Router v7 is now a full-stack framework, offering SSR and all of Remix's advanced features with a flexible setup for both small and large apps

Think of a **restaurant kitchen**.

- **SSR (Server-Side Rendering):**
It's like the chef preparing the entire dish in the kitchen (server) before serving it to you. When the waiter (browser) brings it, the meal is ready to eat. You don't wait for ingredients to arrive one by one; you get the fully cooked plate immediately.
- **CSR (Client-Side Rendering):**
It's like the restaurant giving you raw ingredients (HTML shell + JS bundles) and you have to cook the meal yourself at your table (browser does the rendering). It takes longer before you can actually eat (see the UI).
- **React Router v7 framework mode:**
Here, it's like the kitchen (server) is very smart: it can prepare your dish **based on your exact order (loader + action)**, serve it hot, and also let you customize parts of it at the table if you want (hydration + reactivity on client).

✨ So in **React Router v7 SSR**, the app is like the chef serving you a **ready-to-eat dish (pre-rendered HTML)** quickly, while still letting you **add salt or spice later (client-side interactivity)**.

1. Loader (Chef prepares ingredients before serving)

- **Analogy:** Chef checks the recipe, gathers the right ingredients, and cooks the meal before serving.

```
// routes/products.tsx
```

```
import { type LoaderFunctionArgs } from "react-router";
```

```
export async function loader({ request }: LoaderFunctionArgs) {
```

```
  // Chef prepares data (ingredients)
```

```
  const res = await fetch("https://fakestoreapi.com/products");
```

```
    return res.json();
  }
```

2. SSR (Serve the cooked dish to the customer)

- **Analogy:** The waiter brings you the **fully cooked dish** (HTML rendered with data).

```
// routes/products.tsx
```

```
import { useLoaderData } from "react-router";
```

```
export default function Products() {
```

```
  // Ready-made dish received from kitchen
```

```
  const products = useLoaderData<typeof loader>();
```

```
  return (
```

```
    <div>
```

```
      <h1>🍽️ Product List</h1>
```

```
      <ul>
```

```
        {products.map((p: any) => (
```

```
          <li key={p.id}>{p.title} - ${p.price}</li>
```

```
        )))
```

```
      </ul>
```

```
    </div>
```

```
  );
```

```
}
```

Here, the **server** has already prepared the HTML with the product list before the browser sees it.

3. Hydration (You add spice/sauce at the table)

- **Analogy:** The dish is cooked, but you can still add salt, chili, or sauce (interactivity added on client).
- **Code:**
React Router automatically hydrates the server-rendered HTML, so you can still click buttons, filter products, etc., without a full reload.

```
function ProductFilter() {  
  
  const [filter, setFilter] = React.useState("");  
  
  return (  
  
    <div>  
  
      <input  
  
        placeholder="Filter products"  
  
        value={filter}  
  
        onChange={e => setFilter(e.target.value)}  
  
      />  
  
      { /* spices added here client-side */ }  
  
    </div>  
  
  );  
}
```

Loader = Chef cooking before serving

SSR = Ready-made dish served to you (fast)

Hydration = You adjust seasoning at the table (client-side interactivity)

E-commerce Analogy

Imagine you are shopping in an online store for clothes.

1. Client-Side Rendering (CSR)

- **Analogy:**
You open the store's website, but at first you only see an empty shop with shelves (basic HTML). Then, as you walk around, items are slowly brought in one by one from the warehouse (API calls). You wait longer before you can browse products.
 - **In e-commerce app:**
 - The browser downloads a JS bundle.
 - Products list, cart, etc., load only after JS runs and calls APIs.
 - First load is slower, but navigation between pages feels fast (because data is fetched dynamically).
-

2. Server-Side Rendering (SSR)

- **Analogy:**
When you enter the shop, the shelves are already stocked with products (server cooked the data into HTML before sending). You can immediately start browsing items. If you click "Add to Cart," interactivity still works

because React hydrates it afterward.

- In e-commerce app:
 - The server fetches product list, builds HTML, and sends it.
 - Faster first paint (SEO friendly, good UX).
 - Client-side JS takes over for cart updates, filters, checkout.
-

3. Static Pre-Rendering (SSG)

- **Analogy:**
It's like a printed product catalog that is ready before you even come to the store. Everyone gets the same catalog instantly, but it doesn't change often (unless reprinted). Great if products don't change frequently (e.g., seasonal sale items).
 - In e-commerce app:
 - HTML pages are generated at build time.
 - Example: Homepage, About Us, FAQs, seasonal sale pages.
 - Super fast (just serve static files), but not ideal for frequently changing stock or real-time pricing.
-

Quick Summary in e-commerce language:

- CSR → Empty shop, stock comes in after you arrive.
- SSR → Shop is already stocked when you enter.
- SSG → Pre-printed product catalog handed to you.



Data Loading in React Router v7 Framework Mode

1. Introduction

In React Router v7 framework mode:

- You use **loaders** to fetch data before rendering a route.
 - Loaders can run **on the client** or **on the server** depending on how you configure them.
 - You can also **statically pre-render data** at build time for rarely changing content.
-

2. Client Data Loading

👉 Data is fetched **inside components** with `useEffect` (like classic CSR).

Example: Fetch products on the client after page loads.

```
// routes/products-client.tsx
```

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

export default function ProductsClient() {

  const [products, setProducts] = useState<any[]>([]);


  useEffect(() => {

    fetch("https://fakestoreapi.com/products")

      .then(res => res.json())
```

```

        .then(data => setProducts(data));
    }, []);

    return (
      <div>
        <h1> Client Data Loading</h1>
        {products.length === 0 ? (
          <p>Loading products...</p>
        ) : (
          <ul>
            {products.map(p => <li key={p.id}>{p.title}</li>)}
          </ul>
        )}
      </div>
    );
  }
}

```

First shows **loading UI**, then products appear.

This is like **CSR**.

3. Server Data Loading

👉 Data is fetched **via route loaders**, and React Router renders HTML with the data preloaded.

Example: Fetch products using a loader.

```

// routes/products-server.tsx

import { type LoaderFunctionArgs } from "react-router";
import { useLoaderData } from "react-router";

export async function loader({ request }: LoaderFunctionArgs) {
  const res = await fetch("https://fakestoreapi.com/products");
  return res.json();
}

export default function ProductsServer() {
  const products = useLoaderData<typeof loader>();

  return (
    <div>
      <h1>&img alt="shopping cart icon" data-bbox="195 560 215 575"/> Server Data Loading</h1>
      <ul>
        {products.map((p: any) => <li key={p.id}><p.title></li>)}
      </ul>
    </div>
  );
}

```

Products arrive **with the first HTML render**.

This is like **SSR**.

4. Static Data Loading

👉 Data is pre-fetched at **build time** and served as static files.
Good for pages like "About Us" or "Sale Banner."

Example: Hard-coded or build-time JSON.

```
// routes/sale.tsx

export async function loader() {

  // Pretend this is generated at build time

  return {

    banner: "🔥 Big Diwali Sale - Up to 50% Off!",
    validTill: "2025-10-20"

  };
}

import { useLoaderData } from "react-router";

export default function SalePage() {

  const data = useLoaderData<typeof loader>();

  return (

    <div>

      <h1>{data.banner}</h1>

      <p>Offer valid till: {data.validTill}</p>

    </div>

  );
}
```

Page is basically **pre-baked**.

Similar to **SSG (Static Site Generation)**.

Using Both Loaders (Nested)

👉 You can combine **parent and child loaders**.

For example, fetch product list in parent, and product details in child.

```
// routes/products.tsx
```

```
import { Outlet, useLoaderData } from "react-router";
```

```
export async function loader() {  
  const res = await fetch("https://fakestoreapi.com/products");  
  return res.json();  
}
```

```
export default function ProductsLayout() {  
  const products = useLoaderData<typeof loader>();  
  return (  
    <div>  
      <h1>🛒 Products</h1>  
      <ul>  
        {products.map((p: any) => (  
          <li key={p.id}>  
            <a href={` /products/${p.id}`}>{p.title}</a>  
          </li>  
        ))}  
      </ul>  
    </div>  
  )}
```

```

    </ul>

    <hr />

    <Outlet /> { /* Child route will render here */ }

  </div>

);

}

// routes/products.$id.tsx

import { type LoaderFunctionArgs, useLoaderData } from "react-router";

export async function loader({ params }: LoaderFunctionArgs) {
  const res = await fetch(`https://fakestoreapi.com/products/${params.id}`);
  return res.json();
}

export default function ProductDetails() {
  const product = useLoaderData<typeof loader>();
  return (
    <div>
      <h2>{product.title}</h2>
      <p>{product.description}</p>
      <p> $ {product.price}</p>
    </div>
  );
}

```

}

Parent loader → fetches product list.

Child loader → fetches details only when user clicks a product.

Very efficient since not all data is fetched at once.

Summary with E-commerce:

- **Client Data Loading** → Customer enters empty store → staff brings products slowly.
- **Server Data Loading** → Customer enters and shelves are already stocked.
- **Static Data Loading** → Store has a printed seasonal catalog ready.
- **Using Both Loaders** → Store has a catalog (list) and when you pick an item, staff shows full product details.