Day 24

React's Server Components

@oluwakemi Oluwadahunsi

Introduction

For Day 22, we learned about **Server-Side Rendering (SSR)**, how it works and an example using Next.js for i mplementation.

Server Components in React is an exciting new feature that brings a fresh approach to building web applications.

It allows **certain parts** of your React app to be rendered on the server and sent to the client.

While **React Server Components (RSC)** and **Server-Side Rendering (SSR)** both involve server-side operations in rendering content, they are fundamentally different techniques, designed for distinct use cases.

Let’s clarify the differences to avoid mistaking them for one another before going deeply into Server Components.

@oluwakemi Oluwadahunsi

SSR Overview

First, let’s see again in summary what SSR is about:

- **Purpose:** SSR is used to generate the entire page's content on the server. When a request is made, the server renders React components into static HTML and sends this pre-rendered HTML to the client.

- **Interactivity:** The client receives a fully rendered HTML page, but **it still needs client-side JavaScript for interactivity**. After the initial HTML is loaded, **React hydrates** the app to make it interactive (attach event listeners, etc.).

- **When to Use:** SSR is ideal for cases **where SEO is crucial** or when you need fast initial load times with complete interactivity, such as blog pages, landing pages, or e-commerce sites.

RSC Overview

- **Purpose:** React Server Components are a newer feature that allows you to render **some components** on the server while they remain static on the client, without requiring client-side JavaScript.

- **Interactivity:** Unlike SSR, Server Components are non-interactive by design. They **never execute JavaScript on the client**. If a part of the UI requires interactivity, **you'd combine Server Components with Client Components**.

- **When to Use:** RSC is best for parts of your UI that **don’t need to respond** to user interaction—such as lists of products, blog post content, or static data-heavy views.

Key Differences Between SSR and RSC:

Features	SSR	RSC
Rendering	Entire page or parts rendered on the server and sent as HTML.	Components rendered on the server, sent as HTML, no client-side JS.
Interactivity	Requires client-side JavaScript to hydrate and become interactive.	Static; no client-side JavaScript for these components (no interactivity).
Performance	Faster initial load but requires client-side hydration to be interactive.	Reduces client-side JavaScript load, can improve overall performance.
Client-side Javascript	Sends both HTML and JavaScript to the client.	Sends only HTML; no JavaScript for Server Components.
Use Case	SEO-sensitive or complex dynamic apps needing fast initial load.	Non-interactive, static components like data-heavy pages or static UI.

@o uwakemi Oluwadahunsi

What Are React Server Components?

Now that we understand how Server-Side Rendering and React Server Components differ, let's learn more about React Server Components.

React Server Components (RSC) are a modern feature that allows you to render **some** of your React components entirely on the server **without** sending JavaScript for those components to the client.

This can significantly reduce the amount of client-side JavaScript, improving both page load times and overall performance, especially for static, content-heavy sections of your app.

@oluwakemi Oluwadahunsi

Key Concepts of React Server Components:

1. No Client-Side JavaScript: The components are rendered as HTML on the server, and only the HTML is sent to the client. This means that no JavaScript for these components runs in the browser.

2. Non-Interactive Components: Since there is no JavaScript on the client, Server Components are non-interactive. If a component needs to be interactive (e.g., a button or a form), you would use a Client Component.

3. Reduced Client-Side Bundle Size: By offloading some rendering to the server and avoiding sending JavaScript for non-interactive components, you reduce the size of the JavaScript bundle your users need to download.

@oluwakemi Oluwadahunsi

4. Seamless Mixing of Client and Server Components: You can use Server Components and Client Components together in the same React app. Server Components handle static content, while Client Components manage user interaction.

Example of React Server Components:

Let’s break down a simple example of using a Server Component in React.

Let’s walk through a basic example of how you might implement **Server Components** in a React app.

Step 1: Setting up a Server Component:

React Server Components require a specific environment to work. In a Next.js app, for instance, server-side logic is already well supported.

But for a React Project, we will need an external environment to be able to run our components on the server.

@oluwakemi Oluwadahunsi

Step 2: install necessary dependencies:

```
1 npm install express react react-dom react-router-dom isomorphic-fetch
```

- express: to create a Node.js server to handle the server-side rendering.
- react-dom/server: to render React components to static HTML on the server.
- isomorphic-fetch: to fetch data from the server, both in the server and client environments.

Step 3: Creating a Custom Server using Express

You need a Node.js server to render React Server Components and serve them to the client. Create an express server that will handle the server-side rendering.

@oluwakemi Oluwadahunsi

Create a new server.js file:

```
// server.js
import express from 'express';
import { renderToPipeableStream } from 'react-dom/server';
import App from './src/App';
import { StaticRouter } from 'react-router-dom/server';
import path from 'path';
import fs from 'fs';

const PORT = 3000;
const app = express();

// Serve static files like JS, CSS, etc.
app.use(express.static(path.resolve(__dirname, 'build')));

// Server-side rendering
app.get('*', (req, res) => {
  const stream = renderToPipeableStream(
    <StaticRouter location={req.url}>
      <App />
    </StaticRouter>,
    {
      onShellReady() {
        res.statusCode = 200;
        res.setHeader('Content-type', 'text/html');
        stream.pipe(res);
      },
      onError(err) {
        console.error(err);
        res.statusCode = 500;
        res.send('Server error');
      }
    }
  );
});

app.listen(PORT, () => {
  console.log(`Server is listening on http://localhost:${PORT}`);
});
```

- express.static(): This serves your static client-side files (CSS, JavaScript) from the build folder.
- renderToPipeableStream: This function is used to render your React app as HTML on the server, using streaming (a feature of React 18).
- StaticRouter: This ensures your React routing works on the server, as you can't use BrowserRouter on the server.

Step 4: Set up the React Components (Server Components)

To illustrate how to use Server Components, let’s assume you want to display a list of products fetched from an API or database.

I will be creating a ProductList.server.js (that should be the naming convention):

```jsx
// src/ProductList.server.js

import fetch from 'isomorphic-fetch';

export default async function ProductList() {
  const res = await fetch('https://fakestoreapi.com/products'); // Fetch products from API
  const products = await res.json();

  return (
    <div>
      <h1>Product List</h1>
      <ul>
        {products.map((product) => (
          <li key={product.id}>{product.title}</li>
        ))}
      </ul>
    </div>
  );
}
```

In this example component:

- fetch: We're using isomorphic-fetch to fetch data on the server (this fetch works in both client and server environments).
- async function: This server component fetches data asynchronously and returns the rendered HTML of the product list.

- Server Component: ProductList is a Server Component. It fetches data (a list of products) on the server side, renders it as HTML, and sends this HTML to the client.
- No Client-Side JavaScript: Since this component is rendered entirely on the server, no JavaScript is shipped to the client for it.
- Non-Interactive: The product list is static. If we wanted to add interactivity (e.g., a button to "add to cart"), we would use a Client Component for that part of the UI.

This component now renders a static HTML of a list of products fetched from the server.

Mixing Server and Client Components

You can mix Server and Client Components in the same React tree.

From our example:

```
// App.js

import ProductList from './ProductList.server'; // Server Component
import CartButton from './CartButton.client';   // Client Component

export default function App() {
  return (
    <div>
      <ProductList />   {/* Rendered on the server, static content */}
      <CartButton />    {/* Rendered on the client, interactive */}
    </div>
  );
}
```

- ProductList is a Server Component that renders on the server and sends HTML to the client.

- CartButton is a Client Component, rendered on the client, and allows interactivity (like adding a product to the cart).

@oluwakemi Oluwadahunsi

In Conclusion

React Server Components help improve performance by sending only the necessary HTML to the client and avoiding sending heavy JavaScript.

This is especially useful for static, non-interactive parts of your application, like product lists or blog posts.

However, this setup requires a bit more effort than using a framework like Next.js, which has these features built in, and wouldn’t be needing an external server setup to function.

@oluwakemi Oluwadahunsi

I hope you found this material useful and helpful.

Remember to:

Like

Save for future reference

&

Share with your network, be helpful to someone 👌

@oluwakemi Oluwadahunsi

Hi There!

Thank you for reading through

Did you enjoy this knowledge?

💼 Follow my LinkedIn page for more work-life balancing and Coding tips.

🌐 LinkedIn: Oluwakemi Oluwadahunsi

kodemaven-portfolio.vercel.app