What Are React Server Components, and How Do They Work?

freecodecamp.org/learn/front-end-development-libraries-v9/lecture-react-strategies-and-debugging/what-are-react-server-components-and-how-do-they-work

React Server Components (RSCs) is a new trend that has changed the way React developers approach things. With RSCs, more work shifts to the server, which has a lot of benefits.

Let's take a look at what server components are, how they work, and what led to the introduction of server components.

React Server Components are React components that render exclusively on the server, which sends only the final HTML to the client. This means those components can directly access server-side resources and dramatically reduce the amount of JavaScript sent to the browser.

React apps have traditionally used a "client component" system that handles everything in a typical React app, such as rendering, interactivity, and side effects. The term "client component" was rarely used until the introduction of React server components recently.

But the client component system comes with some drawbacks like large JavaScript bundles and slower initial load times.

React frameworks like Next.js and Gatsby found workarounds to offload some processes to the server in order to fix those problems, but none of them were standardized. If you've used either framework, you've probably heard about `getServerSideProps` and `getServerData`.

Then came React Server Components, which let you run some components entirely on the server so you can do things like data fetching and computation before any code runs in the user's browser.

Server components were first popularized and are readily available in Next.js. Other frameworks like Remix and Gatsby are catching up, and there's an experimental plugin for Vite called `vite-plugin-react-server` which lets you build server components.

So how do server components work?

One of the best ways to demonstrate React Server Components is with data fetching.

In traditional React client components, you let the browser handle API requests. Since data fetching is a side effect, you make that API call in a `useEffect` hook.

It's also good practice to set state variables like loading, data, and error so you can indicate that the data is loading, display the data when it's ready, or display an error in your app.

With React Server Components, you can move the entire component to the server and fetch data there without having to use `useState` or `useEffect`:

```
const Users = async () => {
  const res = await fetch("https://jsonplaceholder.typicode.com/users");
  const users = await res.json();

  return (
    <>
      <h1 className="text-4xl text-center mt-6">Users</h1>
      <ul className="text-center mt-3">
        {users.map((user) => (
          <li key={user.id}>{user.name}</li>
        ))}
      </ul>
    </>
  );
};

export default Users;
```

Because React Server Components only run on the server, you can just fetch data from an API and render just once. Also, since data fetching happens on the server, closer to the source, your app may perform better, especially for people with slow network connections.

One major gotcha is that all the code for server components remains on the server, and doesn't get shipped to the browser. That means you can't use React hooks with them, and they don't have access to Web APIs or browser event listeners. So how can you add interactivity?

In the Next.js app router, all components are server components by default. If you want to add interactivity, you need to mark the component as a client component with the `"use client"` directive.

Let's say you want to make the previous example a client component. Here's how you can do that:

```
"use client";

import { useState, useEffect } from "react";

const Users2 = () => {
  const [status, setStatus] = useState({
    users: [],
    loading: true,
    error: null,
  });

  async function fetchUsers() {
    try {
      const res = await fetch("https://jsonplaceholder.typicode.com/users");
      const data = await res.json();
      setStatus((prevStatus) => ({
        ...prevStatus,
        users: data,
        loading: false,
      }));
    } catch (err) {
      setStatus((prevStatus) => ({
        ...prevStatus,
        error: err.message,
        loading: false,
      }));
    }
  }

  useEffect(() => {
    fetchUsers();
  }, []);

  if (status.loading) {
    return <p>Loading Users...</p>;
  }
  if (status.error) {
    return <p>Error getting users: {status.error}</p>;
  }

  return (
    <>
      <h1 className="text-4xl text-center mt-6">Users</h1>
      <ul className="text-center mt-3">
        {status.users.map((user) => (
          <li key={user.id}>{user.name}</li>
        ))}
      </ul>
    </>
  );
};

export default Users2;
```

If you want to add interactivity like click events, the component also has to be marked as a client component:

```
"use client";

import { useState } from "react";

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

  return (
    <>
      <h1>Counter</h1>
      <button onClick={() => setCount(count + 1)}>Increment</button>
      <h2>{count}</h2>
      <button onClick={() => setCount(count - 1)}>Decrement</button>
    </>
  );
};

export default Counter;
```

If you don't add the use client directive to the component, you get an error with a message that says "You're importing a server component that needs `useState`. This React hook only works in a client component. To fix, mark the file (or its parent) with the `"use client"` directive."

The main benefits that come with React Server Components are that data fetching becomes simpler, the code is easier to read, and client complexity is reduced.

Questions

Where do React Server Components run?

Only in the browser.

On both the client and server.

Entirely on the server.

Correct!

In a Web Worker.

Why don't React Server Components have access to React hooks and Web APIs?

They are rendered before the browser loads.

They run only on the server and are not shipped to the browser.

Correct!

They are restricted by React's rendering cycle.

They are only used for styling purposes.

How would you add interactivity to a component in Next.js when using the app router?

Wrap the component in a `<ClientProvider>` tag.

Use the `withClient()` function from Next.js.

Enable client-side rendering in `next.config.js`.

Mark the component as a client component with the "use client" directive.

Correct!