What is SSR? How do we achieve SSR in React?

In client side rendering, the **browser** downloads the HTML, CSS and JavaScript(bundled js) files from the server and then renders JavaScript and additional content is fetched from the server. This means that when the page loads for the first time, user sees a blank page , so to display the complete content, the downloaded js code must be rendered. After the rendering phase is complete and the app is interactive the main data(products, images, reviews) is fetched from APIs/database using async functions exist in the downloaded js code.

The problem: We do not see the content(text, data from API) of the page in the page source when the page loads for the first time. You'll only see the minimal HTML structure that was initially loaded, without the content of your React components. Why? Again as we said before during the initial page load, the browser downloads the basic HTML(**index.html** file inside the build folder) file from the server containing a div element with an id that serves as the **root** element, along with the **JavaScript bundle** that contains the React code. However, the JavaScript code has not yet executed, and thus, React components have not been rendered into the page. As a result, the HTML source code will not contain contents of the React app. So when using client-side rendering (CSR) with React, the contents of the components, won't be visible in the page source code. This is because React renders the components on the client side, meaning the final HTML content is constructed and injected into the DOM by JavaScript after the initial HTML(index.html) is loaded.

Once the JavaScript bundle (bundled code created by **webpack** after we build our React app) is loaded, browser starts executing the JavaScript code and the components are rendered on the client-side. This is commonly known as the "**hydration**" process. During hydration, event handlers are attached to the DOM elements and any necessary rendering is performed, updating the HTML with the dynamic content and interactivity

As a consequence, the content that appears in the browser's "view source" option will not reflect the rendered React app. Instead, you'll see the initial HTML content with JavaScript references.

Main Problems with Client side rendering

Performance: Users need to download and execute the JavaScript to be able to view the page.

SEO: Web crawlers might not wait long enough for client side waterfalls to finish.

Data Fetching: The data requirements would be dependent on client's network conditions.

With **SSR**, the server renders the react application and when a request is made it sends the **fully rendered HTML page** that has all the contents required such as some data we get from an API or your own database or some initial text you want to exist in the page source. This **complete HTML page** is then sent to the client's browser which can display the entire HTML page with the data in it immediately. The purpose of using SSR is to ensure that the page's essential content is available immediately in the page source unlike using client side rendering where the browser has to render JavaScript code to show some data and fetch more data from API/server. Using **SSR** allows search engines like google to see the content in your web page when it is crawled and put your web site in the ranking depending on the searched keyword.

If the web page includes the searched keyword then google will consider the web page related to that searched keyword so SSR is important for SEO(Search engine optimization).

There are several mechanisms that we need to know to implement SSR in React:

Suspense: A feature in React to delay rendering until data fetching, async operations, code splitting is finished. It allows us to show loading indicators or fallback content while waiting to get the data.

lazy: It enables us to split code for components. So the JS bunde is splitted into smaller chunks allowing to load part of the app only when they are needed rather than loading the entire app upfront. This way we can improve performance. It's important to note that when using **lazy**, you need to wrap the dynamically imported component with a <**Suspense**> component. The purpose of <**Suspense**> is to handle the rendering of fallback content while the dynamic import is in progress.

However, the problem was that Suspense and code-splitting using **React.lazy** were not compatible with SSR yet, until React 18.

renderToNodeStream: Used on the server side to render a React component to a Node.js readable stream so the server sends rendered HTML to the client

Previously before React 18, **renderToNodeStream** is used, it couldn’t wait for data and would buffer the entire HTML content until the end of the stream.

renderToPipeableStream: It can return pipeable Node.js stream. The **renderToPipeableStream** function does not suffer from these limitations of **renderToNodeStream**.This is available in React 18

Example:

You created a commercial web site. Imagine we first show the product images and detail about the product then comes the reviews, footer etc. down below. With **renderToPipeableStream()** we can first send the top part, images, and details about the product and then rest of page can be loaded. This way you can interact with the part that is loaded while the rest is loading. Because after this part is loaded, react injects its script. this stream is writable stream.

With **renderToNodeStream**, first bit of HTML is sent to the client, client will see the content but will not be able to interact with it or execute and JS logic till the streaming is fully done and then the client hydrates(attach event handlers and make the content interactive) it, injects the script. If you use **renderToNodeStream**, you have to finish data fetching for all the pages and then you can start streaming. this stream is a readable stream.

ReactDOMClient.hydrateRoot : As a replacement for the previous **ReactDOM.hydrate**, the React team has introduced a new **ReactDOMClient.hydrateRoot** in React 18. Selective hydration allows React to start hydrating chunks at different timings and priorities on the client. It can start hydrating as soon as chunks of HTML and JS are received, and prioritize a hydration queue of parts that the user interacted with. So if you want to apply SSR, the server already renders the app so the client is only responsible for hydrating the application and this is done using ReactDOM.hydrateRoot instead of ReactDOM.render

This solves the issues that we had previously: having to wait for all JavaScript to load before hydrating can start, and either hydrating the entire application or none of it.

Problems with **hydrate** / **hydrateRoot**:

1) To attach JavaScript event handlers, you MUST put the components inside **hydrateRoot/hydrate** otherwise event handlers will not work such as clicking on a button will not fire a function.

2) If you have routes on the client side then those routes must exist on the server side, eg: So if we only have path for **/users** that means inside server.js you must have a path for /users and if you want to see the main page / there must be also path for main page in the server.js but that means that we also need to server side render the main page and if you don't want to ssr the main page that is a problem. so If you do not want to server side render App.js as the main page but server side render Users.js then that is a problem so the trick is to use render function not hydrate/hydrateRoot function and provide both paths for **/** and /**users** so **render** function renders the components on the client side but from server.js we return already rendered Users.js component therefore we will see the contents of Users when we visit /users

Steps to apply SSR using React:

1) Create a file in the react project directory, you can call it as server.js
2) Install **react-router-dom** package
3) Install **express** package
4) Install these babel packages: **npm install @babel/preset-react @babel/register @babel/preset-env ignore-styles**
5) We then need to use babel, why? The server.js file contains nodejs code not jsx code, Node.js environment does not support JSX syntax, which is a common issue when using server-side rendering with React. To resolve this error, you need to configure Babel to transpile JSX syntax to standard JavaScript that Node.js can understand. So to set Babel, you need to create a file called:

    .babelrc or **babel.config.json** and add this line:

```
{
  "presets": ["@babel/preset-env", "@babel/preset-react"]
}
```

Or another option is to enable Babel transpilation on the fly for Node.js. This is commonly used in development environments especially when using SSR.

require("@babel/register") imports the babel register module which allows you to transpile ES6+ and React JSX code on the fly. When Node.js import files.

ignore: An array of regular expressions defines which file should be ignored by Babel transpilation.

presets: It enables certain language features . **@babel/preset-env** preset is used to transpile modern JavaScript syntax to older versions of JavaScript.
@babel/preset-react is used to transpile JSX syntax to regular JavaScript.

Or another option is to use **webpack**

webpack:

It is a build tool that takes multiple JavaScript files and their dependencies (such as styles, images, and other assets) and **bundles** them together into a single optimized file or a set of files, usually referred to as "bundles." Webpack helps manage and optimize the assets and dependencies in a web application, making it easier to deliver a performant and efficient website or web application to users.