**Client Side Rendering**

In Client-Side Rendering (CSR) only the barebones HTML container for a page is rendered by the server. The logic, data fetching, templating and routing required to display content on the page is handled by JavaScript code that executes in the browser/client.

CSR became popular as a method of building single-page applications. It helped to blur the difference between websites and installed applications.

Client-side rendering (CSR) refers to the process of rendering web content on the client's device (usually in a web browser) rather than on the server. In a traditional web application, the server is responsible for generating the HTML and sending it to the client, which then renders the page. In contrast, with client-side rendering, much of the rendering process occurs on the client side after the initial HTML, CSS, and JavaScript are loaded.

**Key Components of Client Side Rendering**

**1. Initial HTML Load:** The server sends a minimal HTML document to the client, typically containing the basic structure of the page and references to external resources like stylesheets and scripts.

**2. JavaScript Execution:** Once the initial HTML is loaded, client-side rendering relies heavily on JavaScript. JavaScript frameworks and libraries (such as React, Angular, or Vue.js) play a crucial role in handling the rendering logic on the client side.

**3. Dynamic Content Rendering:** The client's browser executes JavaScript code, which dynamically fetches data from the server (usually through API calls) and updates the DOM (Document Object Model) to reflect the current state of the application. This dynamic rendering allows for a more interactive and responsive user experience.

**4. Single Page Applications (SPAs):** Client-side rendering is often associated with Single Page Applications, where the entire application is loaded on the initial request, and subsequent interactions do not require full-page reloads. Instead, only the necessary data is fetched and the relevant portions of the page are updated.

**Advantages of Client Side Rendering**

**1) Interactivity:** CSR allows for a more dynamic and interactive user experience as changes to the content can be made without reloading the entire page.

**2) Improved Performance:** Once the initial page is loaded, subsequent interactions can be faster as only data needs to be fetched and the DOM can be updated locally.

**Disadvantages of Client Side Rendering**

**1) SEO Challenges:** Search engine optimization (SEO) can be more challenging with CSR because search engines may have difficulty crawling and indexing content that is generated dynamically on the client side.

**2) Initial Load Time:** The initial page load might be slower as it requires downloading the necessary JavaScript libraries and making additional requests to fetch data.

**3) Dependency on JavaScript:** Client-side rendering relies heavily on JavaScript, so if a user has JavaScript disabled or if there are issues with JavaScript execution, the application may not function as intended.

**4) Performance**: With client-side rendering, the response time during interactions is greatly improved as there is no round trip to the server. However, for browsers to render content on client-side the first time, they have to wait for the JavaScript to load first and start processing. Thus users will experience some lag before the initial page loads. This may affect the user experience as the size of JS bundles get bigger and/or the client does not have sufficient processing power.

**Improving Client Side Rendering Performance**

Since performance for CSR is inversely proportional to the size of the JavaScript bundle, the best thing we can do is structure our JavaScript code for optimal performance. Following is a list of pointers that could help.

**1) Budgeting JavaScript:** Ensure that you have a reasonably tight JavaScript budget for your initial page loads. An initial bundle of < 100-170KB minified and gzipped is a good starting point. Code can then be loaded on-demand as features are needed

**2) Preloading:** This technique can be used to preload critical resources that would be required by the page, earlier in the page lifecycle. Critical resources may include JavaScript which can be preloaded by including the following directive in the <head> section of the HTML.

**<link rel="preload" as="script" href="critical.js" />**

This informs the browser to start loading the critical.js file before the page rendering mechanism starts. The script will thus be available earlier and will not block the page rendering mechanism thereby improving the performance.

**3) Lazy loading:** With lazy loading, you can identify resources that are non-critical and load these only when needed. Initial page load times can be improved using this approach as the size of resources loaded initially is reduced. For example., a chat widget component would generally not be needed immediately on page load and can be lazy loaded.

**4) Code Splitting:** To avoid a large bundle of JavaScript code, you could start splitting your bundles. Code-Splitting is supported by bundlers like Webpack where it can be used to create multiple bundles that can be dynamically loaded at runtime. Code splitting also enables you to lazy load JavaScript resources.

**5) Application shell caching with service workers:** This technique involves caching the application shell which is the minimal HTML, CSS, and JavaScript powering a user interface. Service workers can be used to cache the application shell offline. This can be useful in providing a native single-page app experience where the remaining content is loaded progressively as needed.

With these techniques, CSR can help to provide a faster Single-Page Application experience with a decent FCP and TTI. Next, we will see what is available at the other end of the spectrum with Server-Side Rendering.