SSR Workflow:

The workflow to achieve SSR is as follows:

1. Server-side: generate HTML from React via renderToString or via one of the renderTo*Stream APIs.
2. Client-side: fetch the HTML and JS as per usual (e.g., HTTP response). The HTML payload generates the initial DOM and loads the client-side React.
3. Client-side: Once the JS loads, call hydrateRoot to inform React about (1) the root DOM node and (2) the root React component. React will "hydrate" the root DOM node (aka reattach event listeners) and then take over subsequent renders.

When server-side React creates HTML it strips the JS out. Reintroducing the JS is conceptually straightforward: it is an act of attaching event listeners to DOM nodes. (This is harder to do than it is to say.)

For example, let's consider a simple button component.

```
// const [x, setX] = useState(0)
<div>value: {x}</div>
<button onClick={() => setX(x + 1)}>Inc</button>
```

The React button.onClick prop tells React to create a button DOM element and attach the function () => setX(x + 1) as a listener callback to the click event. That JS gets stripped when you generate the HTML. For example, if you called renderToStaticMarkup, you'd get the following.

ref.
https://www.gatlin.io/content/react-ssr-server-side-rendering

Technologies and Tools:

- **Next.js** is a React framework for building full-stack web applications. You use React Components to build user interfaces, and Next.js for additional features and optimizations.

Under the hood, Next.js also abstracts and automatically configures tooling needed for React, like bundling, compiling, and more. This allows you to focus on building your application instead of spending time with configuration.

Whether you're an individual developer or part of a larger team, Next.js can help you build interactive, dynamic, and fast React applications.

Feature:

1. **Routing**

A file-system based router built on top of Server Components that supports layouts, nested routing, loading states, error handling, and more.

2. **Rendering**

Client-side and Server-side Rendering with Client and Server Components. Further optimized with Static and Dynamic Rendering on the server with Next.js. Streaming on Edge and Node.js runtimes.

3. **Data Fetching**

Simplified data fetching with async/await in Server Components, and an extended fetch API for request memoization, data caching and revalidation.

4. **Styling**

Support for your preferred styling methods, including CSS Modules, Tailwind CSS, and CSS-in-JS

5. **Optimizations**

Image, Fonts, and Script Optimizations to improve your application's Core Web Vitals and User Experience.

6. **TypeScript**

Improved support for TypeScript, with better type checking and more efficient compilation, as well as custom TypeScript Plugin and type checker.

ref.
https://asperbrothers.com/wp-content/uploads/2021/07/ssr.png
https://nextjs.org/docs

- **Nuxt.js** is a free and open-source framework with an intuitive and extendable way to create type-safe, performant and production-grade full-stack web applications and websites with Vue.js.

We made everything so you can start writing .vue files from the beginning while enjoying hot module replacement in development and a performant application in production with server-side rendering by default.
Nuxt has no vendor lock-in, allowing you to deploy your application anywhere, even to the edge.

Nuxt comes with built-in server-side rendering (SSR) capabilities by default, without having to configure a server yourself, which has many benefits for web applications:

- Faster initial page load time: Nuxt sends a fully rendered HTML page to the browser, which can be displayed immediately. This can provide a faster perceived page load time and a better user experience (UX), especially on slower networks or devices.
- Improved SEO: search engines can better index SSR pages because the HTML content is available immediately, rather than requiring JavaScript to render the content on the client-side.

- Better performance on low-powered devices: it reduces the amount of JavaScript that needs to be downloaded and executed on the client-side, which can be beneficial for low-powered devices that may struggle with processing heavy JavaScript applications.
- Better accessibility: the content is immediately available on the initial page load, improving accessibility for users who rely on screen readers or other assistive technologies.
- Easier caching: pages can be cached on the server-side, which can further improve performance by reducing the amount of time it takes to generate and send the content to the client.

Overall, server-side rendering can provide a faster and more efficient user experience, as well as improve search engine optimization and accessibility.

ref.
https://nuxt.com/docs/getting-started/introduction
https://files.codingninjas.in/article_images/server-side-rendering-in-vue-js-1-1639122317.jpg