# React Data Loading & State Management

## Data Fetching Patterns

### Client-Side Rendering (CSR)

In client-side rendering, server delivers a minimal HTML file with JS bundle (React, Vue, Angular etc). Browser runs JS to build DOM, fetch data (via REST/ GraphQL) and render UI dynamically.

<React uses a virtual DOM to efficiently update actual DOM after data is loaded>

✅Benefits of CSR

1. Great for rich interactive apps (dashboards, SaaS, games)
2. Fast navigation after first load (no full-page refresh, only data state update)
3. Scalable server – server just serves static HTML + JS skeleton and client handles heavy-lifting \*server-less\*
4. Decoupled frontend/ backend – great for security and API handling

❌Drawbacks of CSR

1. Slower first load (for JS bundle to download/ execute and data load if not handled properly \*view optimization for more strategies)
2. SEO challenges (bots may not wait for JS execution)
3. Poor initial performance on edge devices (mobile CPU has to do heavy JS work/ calculations if applicable)
4. Empty page flash (only skeleton shown until data is finally loaded – can be helped with wireframing or lazy loading)

Standard Vite React app is CSR by default,

* Data fetching happens in useEffect() after initial render of HTML/ JS components
* First HTML served as blank template and data “hydration” only happens on client

Pseudo Code:

<https://github.com/Joe-Zhou-Yubin/frontend_resources/blob/main/2.%20Data%20Loading%20%26%20State%20Management/01.%20Data%20Fetching%20Patterns/ReactCSRExample.js>

1. Initial Load -> Browser receives empty index.html + JS bundle
2. Hydration -> React mounts App into root
3. First Render -> shows “Loading…”
4. useEffect triggers -> fetch API data
5. State updates -> React re-renders with product list

More Resources

CSR vs SSG vs SSR and how to choose: <https://appwrite.io/blog/post/csr-ssg-ssr>

Rendering Strategies – Basics of SSR, SSG, CSR & ISR: <https://dev.to/josefine/rendering-strategies-basics-of-ssr-ssg-csr-isr-ll9>

NEXT JS React CSR: <https://nextjs.org/docs/pages/building-your-application/rendering/client-side-rendering>

### Server-Side Rendering (SSR)

With SSR, server renders React to HTML at request time and returns the HTML to browser. Page is immediately viewable (great for perceived performance and SEO), and React hydrates it on client to attach event handlers to make it more interactive (partial rendering).

Key Flow:

1. Request hits server
2. Server fetches data -> renders React to HTML with data
3. Browser gets full HTML
4. React hydrates on client (for events) - <https://18.react.dev/reference/react-dom/client/hydrateRoot>

✅Benefits of SSR

1. Faster first print/ Time to First Byte + better SEO than CSR for dynamic content
2. Great when page data changes per request (eg. Dashboard, real-time inventory, personalized feeds)

❌Drawbacks of SSR

1. Server cost/ complexity of implementation: full/ partial render for every request
2. Hydration pitfalls: mismatch between server and client markup can cause warnings and random errors; hydration works by shifting CPU compute from server to client device

\* Use SSR when content must be fresh and indexable or need personalized HTML immediately on first load.

Prefer SSG/ ISR when content is mostly static or Prefer CSR for highly interactive, behind-auth apps where SEO isn’t priority (behind protected endpoint etc.)

SSR building blocks in React/ Next.js

* React DOM server APIs: <https://react.dev/reference/react-dom/server>
  + renderToPipeableStream (Node streams)
  + renderToReadableStream (Web streams/ edge)
* Next.js Page Router: <https://nextjs.org/docs/pages/api-reference/functions/get-server-side-props>
  + getServerSideProps runs on every request, returned props are rendered server-side
* Next.js App Router: <https://nextjs.org/docs/app/getting-started/fetching-data>
  + Fetch directly in server components; fine-grained caching (cache, revalidate, no-store) and streaming are first-class
* Caching in Server Components
  + Opt into dynamic data with (cache: ‘no-store’) or time-based revalidation (next: {revalidate: N}) (<https://nextjs.org/docs/14/app/building-your-application/data-fetching/fetching-caching-and-revalidating>)
  + Cache rendering work (RSC payload, HTML) at framework layer or CDNs, further reducing server load (<https://nextjs.org/docs/app/guides/caching>)

Pseudocode

AppRouter component: <https://github.com/Joe-Zhou-Yubin/frontend_resources/blob/main/2.%20Data%20Loading%20%26%20State%20Management/01.%20Data%20Fetching%20Patterns/ReactSSR_Nextjs_AppRouter.js>

getServerSideProps: <https://github.com/Joe-Zhou-Yubin/frontend_resources/blob/main/2.%20Data%20Loading%20%26%20State%20Management/01.%20Data%20Fetching%20Patterns/ReactSSR_Nextjs_getServerSideProps.js>

Node Express Streaming: <https://github.com/Joe-Zhou-Yubin/frontend_resources/blob/main/2.%20Data%20Loading%20%26%20State%20Management/01.%20Data%20Fetching%20Patterns/ReactSSR_NodeExpressStreaming.js>

Further Resources

Perils of Hydration: <https://www.joshwcomeau.com/react/the-perils-of-rehydration>

### Static Site Generation (SSG)

### Incremental Static Regeneration (ISR)

## Optimization Strategies

### Lazy Loading (images, components)

### Infinite Scroll

### Pagination

### Debouncing & Throttling for input-heavy UIs

## State Management

### Local component state

### Global app state

### Context API

### Redux

### Zustand

### Recoil

### Tradeoffs between Centralized vs Distributed state