Rendering pattern

---

1. Client-Side Rendering (CSR)

The "Traditional SPA" approach. The server sends a bare-bones HTML file and a large JavaScript bundle. The browser then executes the JS to build the entire UI.

- **Next.js Example:** Using 'use client' at the top of a file or fetching data inside a useEffect hook.
- **Hydration:** The entire app is built and made interactive in one go once the JS is downloaded.
- **SEO:** Poor. Search crawlers see an empty initially.
- **Performance:** High **TTI** (Time to Interactive) and slow **FCP** (First Contentful Paint) because the user sees a blank screen until the JS bundle is parsed.

---

2. Server-Side Rendering (SSR)

The server generates the full HTML for every request. The user gets a meaningful page immediately, but it isn't interactive until the JS "hydrates" it.

- **Next.js Example:** Using getServerSideProps (Pages Router) or dynamic fetching in an App Router page without caching.
- **Pros:** Excellent SEO; fast **FCP**.
- **Cons:** Higher **TTFB** (Time to First Byte) because the server must fetch data and render HTML before sending anything. High server load.
- **When to use:** Highly dynamic, personalized data that must be fresh (e.g., a personalized social media feed).

---

3. Static Site Generation (SSG)

The HTML is generated once at **build time**. This static file is then cached on a CDN and served to all users.

- **Next.js Example:** The default behavior in App Router or using getStaticProps (Pages Router).
- **Performance:** The fastest pattern. **TTFB** is near-zero because the server just serves a file from the edge.
- **SEO:** Perfect. Content is always present in the HTML.
- **When to use:** Content that doesn't change per user (e.g., Blogs, Marketing pages, Documentation).

> **Pro Tip:** In interviews, mention **ISR (Incremental Static Regeneration)**. It allows you to update static pages *after* build time without a full redeploy by revalidating them in the background.

---

4. React Server Components (RSC)

RSC is the "new paradigm." Unlike SSR, which renders the *entire page* to HTML, RSC allows you to render **individual components** on the server.

- **The Key Difference:** RSCs never hydrate. They send a serialized format (not just HTML) to the client, which React uses to update the DOM. This means the JavaScript for that component **never leaves the server**, drastically reducing bundle size.
- **Pros:** Zero-bundle-size components; direct database access within components; streaming (the UI shows up in "chunks").
- **Cons:** No state (useState) or effects (useEffect) in Server Components.
- **When to use:** Data-heavy parts of the UI that don't need interactivity (e.g., a product description or a list of comments).

Comparison Table

Pattern	Data Freshness	SEO	TTFB	TTI	Bundle Size
CSR	Real-time	Low	Fast	Slow	Large
SSR	Real-time	High	Slow	Medium	Large
SSG	Build-time	High	Instant	Fast	Medium
RSC	Variable	High	Fast*	Fast	**Smallest**

**RSC supports streaming, which keeps TTFB fast while data is still being fetched.*

The "Hydration" Problem

In an interview, you must explain **Hydration**. It is the process where React "wakes up" the static HTML sent by the server by attaching event listeners.

1. **The Gap:** Between FCP (seeing content) and TTI (clicking a button), there is a "dead zone" where the site looks ready but is frozen.
2. **The RSC Solution:** By using Server Components for non-interactive parts, you skip hydration for those areas, reducing the "work" the browser has to do.

Interview Strategy: When to Use Which?

When asked to "Design a Dashboard" or "Design E-commerce," use this logic:

1. **Product Page (SEO critical):** Use **SSG + ISR** for the description/images and **CSR** for the "Add to Cart" button.
2. **User Settings (Personalized):** Use **RSC** to fetch the settings and a small **Client Component** for the toggle switches.
3. **Analytics Dashboard:** Use **CSR** since SEO isn't needed and interactivity (charts/filters) is high.