In the **App Router**, SSR is no longer a "special function" like getServerSideProps. Instead, it is the **default behavior** for any component that fetches dynamic data.

The most significant evolution here is **Streaming SSR**, which solves the "blocking" problem found in the Pages Router.

---

1. The Concept: "Server First" Components

In the App Router, every component is a **Server Component** by default. To perform SSR, you simply make your component async and await your data.

If your data fetching is "dynamic" (e.g., it uses cookies, headers, or you tell it not to cache), Next.js automatically renders that component on the server for every request.

---

2. Technical Implementation

Notice how much cleaner this is compared to the Pages Router. The data fetching lives **inside** the component, not in a separate function at the bottom of the file.

```
// app/dashboard/page.tsx

// 1. This component is a Server Component by default
export default async function DashboardPage() {

  // 2. Data fetching happens directly in the component
  // 'no-store' forces this to be SSR (dynamic) rather than SSG (static)
  const res = await fetch('https://api.example.com/stats', {
    cache: 'no-store'
  });
  const stats = await res.json();

  return (
    <main>
      <h1>Dashboard</h1>
      <section>
        <p>Total Users: {stats.users}</p>
        {/* Only interactive parts need to be separate Client Components */}
        <InteractiveChart data={stats.chartData} />
      </section>
    </main>
  );
}
```

---

3. Streaming & Suspense: The "Game Changer"

In the Pages Router, if your API took 5 seconds to respond, the user saw a blank screen for 5 seconds. In the App Router, we use **Streaming**.

1. **Instant Shell:** Next.js immediately sends the layout (Sidebar, Navbar) to the browser.
2. **Loading State:** You wrap your slow component in <Suspense>. Next.js sends a "loading" placeholder (like a skeleton) in the initial HTML.
3. **Chunked Response:** As soon as the data is ready on the server, Next.js "streams" the final HTML for that component over the same connection.
4. **In-place Replacement:** The browser swaps the skeleton with the real content.

---

4. Hydration in App Router SSR

Hydration is much more efficient in the App Router because of **React Server Components (RSC)**.

- **Partial Hydration:** Only components marked with 'use client' are hydrated. The HTML for Server Components is static and "dead" on the client, so React doesn't waste CPU cycles trying to "attach" to them.
- **Zero Bundle Size:** The code used to fetch data and render the Server Components (like heavy Markdown parsers or DB libraries) stays on the server. It is **never** sent to the browser, making the JS bundle significantly smaller.

---

5. Comparison: Pages Router vs. App Router (SSR)

Feature	**Pages Router SSR**	**App Router SSR**
Data Fetching	getServerSideProps	async/await in component
Response Type	Blocking (All or nothing)	**Streaming** (Piece by piece)
Hydration	Full-page hydration	**Selective** (Only Client Components)

Bundle Size	Larger (Includes component logic)	**Smaller** (Logic stays on server)
DX	Separates data from UI	Co-locates data and UI

---

6. Interview Q&A: App Router SSR

Q1: How do you trigger SSR instead of SSG in the App Router?

Answer: SSR is triggered if you:

1. Use fetch(url, { cache: 'no-store' }).
2. Use "Dynamic Functions" like cookies() or headers().
3. Set export const dynamic = 'force-dynamic' at the top of the route.

Q2: Why is Streaming better for System Design?

Answer: It improves **User Perceived Performance**. By showing the Navbar and a loading skeleton immediately, the user feels the app is fast, even if the data takes a moment to load. It also prevents slow API calls from bottlenecking the entire page.

Q3: Can you use useEffect in an SSR App Router page?

Answer: Only if you mark that specific component with 'use client'. Server Components (the ones doing the SSR) do not have a lifecycle and cannot use hooks. You must move interactivity into "leaf" components.

Q4: What is the RSC Payload?

Answer: In addition to HTML, the server sends a special serialized format (the RSC Payload). This contains the rendered result of Server Components and placeholders for where Client Components should be hydrated. This allows React to update the UI without losing client-side state.
