In the **Next.js Pages Router**, SSR is handled through a specific data-fetching function called *getServerSideProps*. This function tells Next.js: *"Do not render this page until you have executed this code on the server and retrieved the necessary data."*

---

1. The Core Mechanism: getServerSideProps

This function runs on **every single request**. It never runs in the browser. Because it stays on the server, you can perform secure operations like querying a database directly or using private API keys without exposing them to the client.

The Execution Flow

1. **Request:** A user clicks a link or types a URL.
2. **Server Execution:** The server calls getServerSideProps.
3. **Data Retrieval:** The function fetches data (e.g., from an external API or Database).
4. **Render:** Next.js passes this data as props to your React component and renders the component into an HTML string.
5. **Response:** The server sends the complete HTML + the JSON data (the "props") to the browser.
6. **Hydration:** The browser displays the HTML immediately. Then, React uses the JSON "props" to hydrate the page and make it interactive.

---

2. Technical Implementation Detail

In the Pages Router, you export getServerSideProps from your page file.

```
// pages/user/[id].js

// This function runs on the server for EVERY request
export async function getServerSideProps(context) {
  const { params, req, res, query } = context;
  const { id } = params;

  // 1. Direct DB call or Secure API call
  const response = await fetch(`https://api.privatedata.com/users/${id}`, {
    headers: { Authorization: process.env.PRIVATE_TOKEN }
  });
  const userData = await response.json();
  // 2. Return the data as props
  return {
    props: {
      user: userData
    }
  };
}

// The component receives the data before it even reaches the browser
export default function UserProfile({ user }) {
  return (
    <div>
      <h1>{user.name}</h1>
      <p>{user.bio}</p>
    </div>
  );
}
```

---

3. Hydration in Pages Router SSR

Hydration in the Pages Router is **top-down and total**.

- **The Payload:** The server sends the HTML, but it also embeds a script tag containing the data in a JSON format (usually inside a __NEXT_DATA__ script tag).
- **The Process:** React reads that JSON on the client. It rebuilds the Virtual DOM using that data and ensures it matches the HTML sent by the server.
- **The Result:** If it matches, React attaches event listeners. If it doesn't match (e.g., if you used Math.random() in the component), you get a **Hydration Error**.

---

4. Performance & System Design Trade-offs

The "Blocking" Problem

The biggest drawback of SSR in the Pages Router is that **it is blocking**. The browser will show a "loading" spinner in the tab (or a blank screen) while the server is waiting for getServerSideProps to finish. If your API is slow, your user sees nothing.

Metrics Impact

- **TTFB (Time to First Byte):** High (Slower). The server has to "do work" before sending the first byte.
- **FCP (First Contentful Paint):** Fast. Once the byte arrives, it contains the full UI.
- **LCP (Largest Contentful Paint):** Generally good, as the main content is in the initial HTML.

---

5. Interview Q&A: SSR (Pages Router)

Q1: What happens if getServerSideProps fails?

Answer: You can return a notFound: true object to trigger a 404 page, or a redirect object to send the user to a login page. This allows for server-side access control.

Q2: Does SSR code increase the client-side bundle size?

Answer: No. Code inside getServerSideProps and any modules imported **only** for use inside that function are stripped from the client-side JavaScript bundle by Next.js. This is a major benefit for performance.

Q3: When should you prefer SSR over SSG (Static Site Generation)?

Answer: Use SSR when the data is **dynamic and user-specific** (like a personalized dashboard) or when the data changes so frequently that a static build would be outdated immediately. If the data is the same for everyone (like a blog post), use SSG.

Q4: How do you optimize SSR in a high-traffic system?

Answer:

1. **Caching:** Use a stale-while-revalidate cache at the CDN level using Cache-Control headers.
2. **Database Optimization:** Ensure the queries inside getServerSideProps are indexed and fast.
3. **Parallel Fetching:** If you need data from multiple APIs, use Promise.all() to fetch them concurrently rather than sequentially.
