Engineer

Server-side rendering

producing a page's HTML on the server for each request

What it is

Server-side rendering, usually shortened to SSR, means producing a page's HTML on the server at the moment someone asks for it. The browser receives a complete document with the content already in place and can show it before any script has run. In a JavaScript framework the same components then run a second time in the browser, which takes over the HTML it was sent and makes it interactive.

The term answers a question about timing: when, and where, is the HTML made? Three answers are common. A static site generator makes it once, at build time, and serves the same file to everyone. Server-side rendering makes it on the server for each request, so the page can depend on who is asking and on what the data says at that moment. Client-side rendering, often called CSR, sends a nearly empty document and a bundle of JavaScript that builds the page in the browser. Real applications often mix all three, one route at a time.

Next.js puts it the same way. Its documentation gives dynamic rendering as another name for SSR and says that the page's HTML is generated on each request. In the Pages Router a page opts in by exporting an async function called `getServerSideProps`, which the server calls on every request; its sibling `getStaticProps` does the same job once, at build time. React supplies the machinery underneath. Its `react-dom/server` APIs turn a tree of components into HTML on the server, either as a stream, with `renderToPipeableStream` for Node.js and `renderToReadableStream` for Web Streams, or in one piece with the older `renderToString`, which React lists as a legacy API with limited functionality.

The second half happens in the browser and is called hydration. Next.js describes the first load this way: the HTML gives a fast, non-interactive preview, and JavaScript then hydrates the components, which it defines as React connecting

the event handlers to the existing DOM so the page responds to input. React's reference for `hydrateRoot` is frank about the trade. The HTML snapshot makes the app appear to load sooner, and the page looks ready a moment before it is.

Use the concept in a decision

Put "What is server-side rendering?" to work in a real decision. We can help relate the concept to the system you are building or buying. Interfaces that load quickly, work on a phone and behave the same in every browser your customers happen to use.

Explore our services Get in touch →

Test the assumptions behind the term

Use the explanation and linked work to frame questions for your project. We can examine the assumptions with you before a tool is chosen. Directed full-stack GIS map development — full-stack mapping, from database to browser

Directed full-stack GIS map development Get in touch →

Why it matters

The content is in the first response. A reader on a slow connection sees text and images while the JavaScript is still on its way, and a crawler that does not run JavaScript sees the same page a person does rather than an empty shell. The browser does not have to run a script and then fetch data before it can paint real content.

The page can also depend on the request, which is what sets it apart from a static build. Next.js recommends `getServerSideProps` for data that is personal to the user or known only at request time, such as an authorization header or the visitor's location, and points everything else towards build-time generation. The data is fetched next to where it lives. Code inside the function runs only on the server, its imports are left out of the client bundle, and it can query a database or a content management system directly. One trip to the server can replace the requests a browser would otherwise make one after another once its scripts had loaded.

The costs mirror a static site's savings. There has to be a running server, and it does the rendering work again for every request. The response can start only once the data is there, so without streaming the slowest query sets the pace for the whole page. And server rendering does not remove the JavaScript: the browser still downloads every component it has to hydrate, and until hydration finishes, a button can be drawn on the screen and still do nothing.

That makes it a choice per page rather than per project. A page that is the same for every reader gains little from being rendered again on every request, and a page that changes with the reader, or with data that moves by the minute, is where SSR earns its server. This site sits at the static end. Its tech stack is static-first where that fits, so these pages need no request-time application execution, and Hugo builds the site.

What it looks like in practice

The page below, written for this article, uses the Pages Router. It lists a signed-in customer's orders, sends a visitor without a session to the sign-in page and tells every cache not to keep the result. `readSession` and `db` stand for whatever the application uses for sessions and data.

```
// pages/orders.js
export async function getServerSideProps({ req, res }) {
  const session = await readSession(req.cookies.session);
  if (!session) {
    return { redirect: { destination: "/sign-in", permanent: false } };
  }

  const rows = await db.orders.listFor(session.userId);
  res.setHeader("Cache-Control", "no-store");

  return {
    props: {
      orders: rows.map((o) => ({
        id: o.id,
        number: o.number,
        status: o.status,
      })),
    },
  };
}

export default function Orders({ orders }) {
  return (
    <ul>
      {orders.map((o) => (
        <li key={o.id}>
          {o.number}: {o.status}
        </li>
      ))}
    </ul>
  );
}
```

Next.js runs the function on every request to `/orders`. `req.cookies` is the request's cookie header, already parsed. A `redirect` with `permanent: false` is a temporary redirect rather than a permanent one. Whatever comes back under `props` has to survive `JSON.stringify`, and it reaches the browser too, as part of the initial HTML, so that React can hydrate the list with the same data the server used. That is why the function maps each row down to the three fields the page shows instead of passing the database rows straight through. When a reader later arrives through a `next/link` link, Next.js asks the server to run the function again and gets the props back as JSON, without a full page load.

In the founder's current work under NextMariner, the Next.js 16 frontend shows the problem this solves. Its portfolio entry starts from an authenticated dashboard that had to feel quick and stay interactive. Fetching everything in the browser makes the first load drag and brings the N+1 pattern with it: each component wakes up and sends its own request, so a single page turns into a cascade of round trips. On that dashboard and on the large directory listings, a factory called `createPrefetchedServerPage` now fetches the page's data on the server and hands it to the client already populated, so the components start with their data instead of each going off to ask for it. Pages moved onto the factory come up in one round trip instead of a storm of them. The same entry keeps the other two modes where they suit: public pages are generated statically because they do not change from user to user, and the genuinely interactive parts stay client-rendered. It also documents a cache-invalidation strategy for each data request, so the data stays fresh without fetching again what the app already has.

Where it goes wrong

The server and the browser disagree. Hydration expects the browser's first render to produce the same output as the server's. A component that checks `typeof window`, calls `window.matchMedia` while rendering or prints the current time renders one thing on the server and another in the browser. React's list of causes also includes different data on the two sides and stray whitespace inside the root element. At best a mismatch slows hydration down, and at worst event handlers end up attached to the wrong elements, so its reference for `hydrateRoot` says plainly: "You should treat mismatches as bugs and fix them." Render the same thing on both sides. Where the browser really must show something else, render the server's version first and switch in an effect after hydration, which costs a second render, and keep `suppressHydrationWarning` for an unavoidable difference such as a timestamp, remembering that it works only one level deep.

Secrets ride along in the props. Everything returned as props is written into the initial HTML so that the browser can hydrate, and the Next.js documentation warns against passing anything sensitive this way. A function that returns a whole user record hands its password hash and internal flags to anyone who opens the page source. Return only the fields the page displays, as the example above does.

A shared cache serves one reader's page to the next. The Next.js documentation illustrates caching for server-rendered pages with `Cache-Control: public, s-maxage=10, stale-while-revalidate=59`. Copied onto a personalised page, that header invites a content delivery network or a proxy in front of the server to keep the response and hand it to whoever asks next. RFC 9111 defines `public` as permission for a cache to store a response it would otherwise refuse, and `s-maxage` as a lifetime meant for shared caches; either one also lifts the rule that stops a shared cache from reusing a response to a request that carried an `Authorization` header. A page built from cookies or credentials needs `private`, which forbids shared caches to store it, or `no-store`, which forbids every cache.

The slowest query holds up the page. Rendered in one piece, the page sends nothing until every query behind it has returned, so one slow query delays the header, the navigation and everything else. React's streaming renderer splits the page at `<Suspense>` boundaries. It sends the shell, everything outside any boundary, as soon as that is ready, with fallbacks in place, then streams each remaining part with a small inline script that swaps it in. One catch follows from sending early: once streaming starts, the status code can no longer change, so a missing record or a failed check has to be settled before the shell goes out, and `onShellError` is where an error status is set.

Further reading

- Server-side Rendering (SSR) — Next.js on HTML generated for each request and how `getServerSideProps` differs from `getStaticProps`.
- getServerSideProps — when to use it, what reaches the client in the initial HTML, and caching with `Cache-Control`.
- Server React DOM APIs — the streaming and legacy APIs React offers for rendering components to HTML on the server.
- hydrateRoot — how React takes over server-rendered HTML, and the common causes of hydration mismatches.
- renderToPipeableStream — the shell, `<Suspense>` boundaries and the status-code caveat of streaming.

- RFC 9111: HTTP Caching — the `public`, `private`, `s-maxage` and `no-store` directives and the rules for shared caches.

Glossary

Choose for your actual constraints

Describe the workflow, the existing interfaces and the users who need the change. We can define a testable delivery and a handover that lets your team maintain it.

See what it costs Get in touch →

Make the terminology practical

Ask how “What is server-side rendering?” applies to your organisation. A concrete example from your current system is enough to start the discussion.

Explore our services

welcome@engineer.company +45 50 10 61 84 Copenhagen, Denmark

Take this page with you

Contact card

Carefully Crafted With Love by D.

CVR 45969703 · Tagensvej 105C, 3. th, 2200 København N

Copyright © 2025 – present · Engineer ApS · All rights reserved.

Printed from https://engineer.company/glossary/server-side-rendering/