Engineer

serverseitiges Rendering

Seiten-HTML für jede Anfrage auf dem Server erzeugen

Was es ist

Serverseitiges Rendering, meist zu SSR abgekürzt, bedeutet, das HTML einer Seite genau dann auf dem Server zu erzeugen, wenn jemand sie anfordert. Der Browser erhält ein vollständiges Dokument mit bereits eingefügten Inhalten und kann es anzeigen, bevor irgendein Skript gelaufen ist. In einem JavaScript-Framework laufen dieselben Komponenten anschließend ein zweites Mal im Browser. Er übernimmt das empfangene HTML und macht es interaktiv.

Der Begriff beantwortet eine Frage nach Zeitpunkt und Ort: Wann und wo entsteht das HTML? Drei Antworten sind üblich. Ein statischer Websitegenerator erzeugt es einmal beim Build und liefert allen dieselbe Datei. Serverseitiges Rendering erzeugt es auf dem Server für jede Anfrage. Die Seite kann deshalb davon abhängen, wer sie anfordert und was die Daten in diesem Moment aussagen. Clientseitiges Rendering, oft CSR genannt, sendet ein nahezu leeres Dokument und ein JavaScript-Bündel, das die Seite im Browser aufbaut. Echte Anwendungen mischen häufig alle drei Verfahren und entscheiden für jede Route einzeln.

Next.js beschreibt es ebenso. Seine Dokumentation nennt dynamisches Rendering als weiteren Namen für SSR und erklärt, dass das HTML der Seite bei jeder Anfrage erzeugt wird. Im Pages Router wählt eine Seite das Verfahren durch den Export einer asynchronen Funktion namens `getServerSideProps`, die der Server bei jeder Anfrage aufruft. Ihr Gegenstück `getStaticProps` erledigt dieselbe Aufgabe einmal beim Build. Die technische Grundlage liefert React. Seine APIs unter `react-dom/server` verwandeln einen Komponentenbaum auf dem Server in HTML: als Stream mit `renderToPipeableStream` für Node.js und `renderToReadableStream` für Web Streams oder in einem Stück mit dem älteren `renderToString`, das React als Legacy-API mit eingeschränktem Funktionsumfang aufführt.

Die zweite Hälfte geschieht im Browser und heißt Hydration. Next.js beschreibt das erste Laden so: Das HTML liefert eine schnelle, noch nicht interaktive Vorschau; anschließend hydriert JavaScript die Komponenten. Das bedeutet laut Definition, dass React die Ereignisbehandler mit dem vorhandenen DOM verbindet, damit die Seite auf Eingaben reagiert. Die React-Referenz zu `hydrateRoot` benennt den Zielkonflikt offen. Der HTML-Schnappschuss lässt die Anwendung früher geladen erscheinen, und die Seite wirkt schon bereit, kurz bevor sie es tatsächlich ist.

Das Konzept für eine Entscheidung nutzen

Nutzen Sie „Was ist serverseitiges Rendering?“ für eine konkrete Entscheidung. Wir können helfen, das Konzept mit dem System in Verbindung zu bringen, das Sie entwickeln oder einkaufen. Oberflächen, die schnell laden, auf dem Smartphone funktionieren und sich in allen Browsern Ihrer Kunden gleich verhalten.

Unsere Leistungen entdecken Kontakt aufnehmen →

Die Annahmen hinter dem Begriff prüfen

Nutzen Sie die Erklärung und die verlinkten Arbeiten, um Fragen für Ihr Projekt zu formulieren. Wir können die Annahmen gemeinsam mit Ihnen prüfen, bevor ein Werkzeug ausgewählt wird. Full-Stack-GIS-Kartenentwicklung geleitet — Full-Stack-Kartenentwicklung von der Datenbank bis zum Browser

Full-Stack-GIS-Kartenentwicklung geleitet Kontakt aufnehmen →

Warum es wichtig ist

Die Inhalte stehen in der ersten Antwort. Wer eine langsame Verbindung hat, sieht Text und Bilder, während das JavaScript noch unterwegs ist. Ein Crawler, der kein JavaScript ausführt, sieht dieselbe Seite wie ein Mensch statt einer leeren Hülle. Der Browser muss nicht erst ein Skript ausführen und danach Daten abrufen, bevor er echte Inhalte zeichnen kann.

Die Seite kann außerdem von der Anfrage abhängen. Das unterscheidet sie von einem statischen Build. Next.js empfiehlt `getServerSideProps` für benutzerspezifische oder erst zum Anfragezeitpunkt bekannte Daten, etwa einen Autorisierungsheader oder den Standort des Besuchers, und verweist für alles andere auf die Erzeugung beim Build. Der Abruf geschieht nahe am Speicherort der Daten. Code innerhalb der Funktion läuft nur auf dem Server, seine Importe gelangen nicht in das Client-Bündel, und er kann eine Datenbank oder ein Content-Management-System direkt abfragen. Ein einziger Anfrage-Antwort-Zyklus mit dem Server kann die Anfragen ersetzen, die ein Browser nach dem Laden seiner Skripte sonst nacheinander stellen würde.

Die Kosten sind das Gegenstück zu den Einsparungen einer statischen Website. Ein Server muss laufen und die Renderingarbeit bei jeder Anfrage erneut ausführen. Die Antwort kann erst beginnen, wenn die Daten vorliegen. Ohne Streaming gibt deshalb die langsamste Abfrage das Tempo für die gesamte Seite vor. Serverseitiges Rendering beseitigt auch nicht das JavaScript: Der Browser lädt weiterhin jede zu hydrierende Komponente herunter. Bis die Hydration abgeschlossen ist, kann eine Schaltfläche bereits zu sehen sein und trotzdem noch nichts tun.

Die Entscheidung fällt deshalb pro Seite statt pro Projekt. Eine für alle Leser gleiche Seite gewinnt wenig dadurch, bei jeder Anfrage neu gerendert zu werden. SSR rechtfertigt den Serveraufwand dort, wo sich die Seite mit dem Leser oder mit minütlich wechselnden Daten ändert. Diese Website liegt am statischen Ende. Ihr Technologie-Stack setzt auf statische Seiten, wo das passt: Für diese Seiten muss bei einer Anfrage kein Anwendungscode ausgeführt werden. Hugo baut die Website.

Wie es in der Praxis aussieht

Die folgende, für diesen Artikel geschriebene Seite verwendet den Pages Router. Sie listet die Bestellungen eines angemeldeten Kunden auf, leitet Besucher ohne Sitzung zur Anmeldung weiter und weist jeden Cache an, das Ergebnis nicht zu speichern. `readSession` und `db` stehen stellvertretend für die Sitzungs- und Datenverwaltung der jeweiligen Anwendung.

```
// 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 führt die Funktion bei jeder Anfrage an `/orders` aus. `req.cookies` ist der bereits geparste Cookie-Header der Anfrage. Ein `redirect` mit `permanent: false` ist eine vorübergehende statt einer dauerhaften Weiterleitung. Alles unter `props` muss sich mit `JSON.stringify` serialisieren lassen und gelangt als Teil des anfänglichen HTML auch in den Browser. So kann React die Liste mit denselben Daten hydrieren, die der Server verwendet hat. Deshalb reduziert die Funktion jede Zeile auf die drei angezeigten Felder, statt die Datenbankzeilen unverändert durchzureichen. Kommt ein Leser später über einen Link mit `next/link`, lässt Next.js die Funktion erneut auf dem Server ausführen und erhält die Props als JSON, ohne die ganze Seite neu zu laden.

In der aktuellen Arbeit der Gründerin für NextMariner zeigt das Next.js-16-Frontend, welches Problem damit gelöst wird. Sein Portfolioeintrag beginnt mit einem Dashboard hinter einer Anmeldung, das sich schnell anfühlen und interaktiv bleiben sollte. Alles im Browser abzurufen verzögert das erste Laden und bringt das N+1-Muster mit: Jede Komponente wird aktiv und sendet ihre eigene Anfrage, sodass aus einer Seite eine Kaskade von Anfragen und Antworten wird. Auf diesem Dashboard und in den großen Verzeichnislisten ruft eine Factory namens `createPrefetchedServerPage` die Seitendaten inzwischen auf dem Server ab und übergibt sie dem Client bereits vollständig geladen. Die Komponenten beginnen mit ihren Daten, statt sie jeweils selbst anzufordern. Auf die Factory umgestellte Seiten laden in einem einzigen Anfrage-Antwort-Zyklus statt in einer Flut davon. Derselbe Eintrag behält die beiden anderen Verfahren dort bei, wo sie passen: Öffentliche Seiten werden statisch erzeugt, weil sie sich nicht von Benutzer zu Benutzer unterscheiden, und die tatsächlich interaktiven Teile bleiben clientseitig gerendert. Er dokumentiert außerdem für jede Datenanfrage eine Strategie zur Cache-Invalidierung. So bleiben die Daten aktuell, ohne bereits in der Anwendung Vorhandenes erneut abzurufen.

Wo es schiefgeht

Server und Browser liefern unterschiedliche Ergebnisse. Die Hydration erwartet, dass das erste Rendering im Browser dieselbe Ausgabe wie das des Servers erzeugt. Eine Komponente, die `typeof window` prüft, beim Rendern `window.matchMedia` aufruft oder die aktuelle Uhrzeit ausgibt, rendert auf dem Server etwas anderes als im Browser. React nennt als weitere Ursachen unterschiedliche Daten auf beiden Seiten und zusätzliche Leerzeichen innerhalb des Wurzelelements. Im besten Fall verlangsamt eine Abweichung die Hydration; im schlimmsten Fall landen Ereignisbehandler an den falschen Elementen. Die Referenz zu `hydrateRoot` sagt deshalb ausdrücklich: „Sie sollten Abweichungen als Fehler behandeln und beheben." Rendern Sie auf beiden Seiten dasselbe. Muss der Browser tatsächlich etwas anderes zeigen, rendern Sie zuerst die Serverversion und wechseln Sie nach der Hydration in einem Effect. Das kostet ein zweites Rendering. Setzen Sie `suppressHydrationWarning` nur für unvermeidbare Unterschiede wie einen Zeitstempel ein und beachten Sie, dass es nur eine Ebene tief wirkt.

Secrets gelangen mit den Props nach draußen. Alles, was als Props zurückkommt, wird in das anfängliche HTML geschrieben, damit der Browser hydrieren kann. Die Next.js-Dokumentation warnt davor, auf diesem Weg sensible Daten zu übergeben. Eine Funktion, die einen vollständigen Benutzerdatensatz zurückgibt, liefert dessen Passwort-Hash und interne Kennzeichen an jeden aus, der den Seitenquelltext öffnet. Geben Sie wie im obigen Beispiel nur die Felder zurück, die die Seite anzeigt.

Ein gemeinsamer Cache liefert die Seite eines Lesers an den nächsten aus. Die Next.js-Dokumentation zeigt das Caching serverseitig gerenderter Seiten anhand von `Cache-Control: public, s-maxage=10, stale-while-revalidate=59`. Auf eine personalisierte Seite übertragen erlaubt dieser Header einem Content Delivery Network oder einem vorgeschalteten Proxy, die Antwort zu speichern und dem nächsten Anfragenden auszuliefern. RFC 9111 definiert `public` als Erlaubnis für einen Cache, eine sonst abgelehnte Antwort zu speichern, und `s-maxage` als Lebensdauer für gemeinsame Caches. Jede der beiden Direktiven hebt außerdem die Regel auf, nach der ein gemeinsamer Cache eine Antwort auf eine Anfrage mit `Authorization`-Header nicht wiederverwenden darf. Eine aus Cookies oder Zugangsdaten aufgebaute Seite braucht `private`, das gemeinsamen Caches das Speichern untersagt, oder `no-store`, das es jedem Cache untersagt.

Die langsamste Abfrage hält die Seite auf. Wird die Seite in einem Stück gerendert, sendet sie nichts, bis alle dahinterliegenden Abfragen geantwortet haben. Eine langsame Abfrage verzögert so Kopfbereich, Navigation und alles andere. Reacts Streaming-Renderer teilt die Seite an den Grenzen von `<Suspense>`. Er sendet das Grundgerüst, also alles außerhalb dieser Grenzen, sobald es bereit ist, und setzt dabei vorläufige Inhalte ein. Anschließend überträgt er jeden übrigen Teil mit einem kleinen Inline-Skript, das ihn anstelle des vorläufigen Inhalts einsetzt. Das frühe Senden hat eine Folge: Sobald das Streaming beginnt, kann sich der Statuscode nicht mehr ändern. Ein fehlender Datensatz oder eine fehlgeschlagene Prüfung muss deshalb vor dem Versand des Grundgerüsts geklärt sein; in `onShellError` wird ein Fehlerstatus gesetzt.

Weiterführende Lektüre

- Server-side Rendering (SSR) — Next.js über bei jeder Anfrage erzeugtes HTML und den Unterschied zwischen `getServerSideProps` und `getStaticProps`.
- getServerSideProps — Einsatzfälle, die im anfänglichen HTML an den Client gelangenden Daten und Caching mit `Cache-Control`.
- Server React DOM APIs — Die Streaming- und Legacy-APIs von React zum serverseitigen Rendern von Komponenten in HTML.
- hydrateRoot — Die Übernahme serverseitig gerenderten HTMLs durch React und häufige Ursachen von Abweichungen bei der Hydration.
- renderToPipeableStream — Grundgerüst, Grenzen von `<Suspense>` und der Vorbehalt zum Statuscode beim Streaming.
- RFC 9111: HTTP Caching — Die Direktiven `public`, `private`, `s-maxage` und `no-store` sowie die Regeln für gemeinsame Caches.

Glossar

Für Ihre tatsächlichen Rahmenbedingungen auswählen

Beschreiben Sie den Arbeitsablauf, die vorhandenen Schnittstellen und die Nutzer, die die Veränderung benötigen. Wir können eine prüfbare Lieferung definieren und die Übergabe so gestalten, dass Ihr Team die gelieferten Bestandteile warten kann.

Preise ansehen Kontakt aufnehmen →

Die Begriffe praktisch nutzbar machen

Fragen Sie, wie „Was ist serverseitiges Rendering?“ auf Ihre Organisation zutrifft. Ein konkretes Beispiel aus Ihrem bestehenden System reicht aus, um das Gespräch zu beginnen.

Unsere Leistungen entdecken

welcome@engineer.company +45 50 10 61 84 Kopenhagen, Dänemark

Nehmen Sie diese Seite mit

Kontaktkarte

Mit Sorgfalt und Liebe gestaltet von D.

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

Copyright © 2025 – heute · Engineer ApS · Alle Rechte vorbehalten.

Gedruckt von https://engineer.company/de/glossary/server-side-rendering/