Engineer

Server-side rendering

å lage en sides HTML på serveren for hver forespørsel

Hva det er

Server-side rendering, som regel forkortet til SSR, betyr at HTML-en til en side lages på serveren i det øyeblikket noen ber om den. Nettleseren får et ferdig dokument med innholdet på plass og kan vise det før noe skript har kjørt. I et JavaScript-rammeverk kjører de samme komponentene deretter en gang til i nettleseren, som tar over HTML-en den fikk tilsendt, og gjør den interaktiv.

Begrepet svarer på et spørsmål om tidspunkt: når, og hvor, blir HTML-en laget? Tre svar er vanlige. En statisk nettstedsgenerator lager den én gang, når nettstedet bygges, og sender den samme filen til alle. Server-side rendering lager den på serveren ved hver forespørsel, slik at siden kan avhenge av hvem som spør, og av hva dataene sier akkurat da. Klientside-rendering, ofte kalt CSR, sender et nesten tomt dokument og en pakke JavaScript som bygger siden i nettleseren. Virkelige applikasjoner blander ofte alle tre, én rute om gangen.

Next.js sier det på samme måte. Dokumentasjonen gir dynamic rendering som et annet navn på SSR og sier at HTML-en til siden genereres ved hver forespørsel. I Pages Router velger en side dette ved å eksportere en asynkron funksjon som heter `getServerSideProps`, og som serveren kaller ved hver forespørsel. Søsterfunksjonen `getStaticProps` gjør den samme jobben én gang, når nettstedet bygges. React står for maskineriet under. API-ene i `react-dom/server` gjør et tre av komponenter om til HTML på serveren, enten som en strøm, med `renderToPipeableStream` for Node.js og `renderToReadableStream` for Web Streams, eller i ett stykke med den eldre `renderToString`, som React oppfører som et legacy-API med begrenset funksjonalitet.

Den andre halvdelen skjer i nettleseren og kalles hydrering. Next.js beskriver den første innlastingen slik: HTML-en gir en rask, ikke-interaktiv forhåndsvisning, og deretter hydrerer JavaScript komponentene. Hydrering definerer Next.js som

Reacts måte å feste hendelseshåndterere til DOM-en på, slik at den statiske HTML-en blir interaktiv. Reacts referanse for `hydrateRoot` legger ikke skjul på avveiningen. Øyeblikksbildet i HTML-en får appen til å se ut som om den lastes raskere, og siden ser ferdig ut et øyeblikk før den er det.

Bruk begrepet i en beslutning

Bruk «Hva er server-side rendering?» i en konkret beslutning. Vi kan knytte begrepet til systemet dere bygger eller kjøper. Grensesnitt som laster raskt, fungerer på mobilen og oppfører seg likt i alle nettlesere kundene dine måtte bruke.

Se tjenestene våre Ta kontakt →

Undersøk antakelsene bak ordet

Bruk forklaringen og det tilknyttede arbeidet til å formulere spørsmål for prosjektet. Vi kan undersøke antakelsene sammen før et verktøy velges. Styrte fullstack-utviklingen av GIS-kart — fullstack-kartutvikling, fra database til nettleser

Styrte fullstack-utviklingen av GIS-kart Ta kontakt →

Hvorfor det er viktig

Innholdet er med i det første svaret. En leser på en treg forbindelse ser tekst og bilder mens JavaScript fortsatt er på vei, og en søkerobot som ikke kjører JavaScript, ser den samme siden som et menneske og ikke et tomt skall. Nettleseren trenger ikke først å kjøre et skript og så hente data før den kan tegne ekte innhold.

Siden kan også avhenge av forespørselen, og det er nettopp det som skiller den fra et statisk bygg. Next.js anbefaler `getServerSideProps` for data som er personlige for brukeren eller først blir kjent ved forespørselen, for eksempel en authorization-header eller den besøkendes plassering, og viser alt annet til generering når nettstedet bygges. Dataene hentes nær der de bor. Koden i funksjonen kjører bare på serveren, importene holdes utenfor klientens bundle, og den kan spørre en database eller et CMS direkte. Én tur til serveren kan erstatte forespørslene en nettleser ellers ville sendt etter hverandre når skriptene var lastet.

Kostnadene er speilbildet av det et statisk nettsted sparer. Det må kjøre en server, og den gjør renderingsarbeidet på nytt ved hver forespørsel. Svaret kan først begynne når dataene er på plass, så uten strømming bestemmer det tregeste dataoppslaget tempoet for hele siden. Og rendering på serveren fjerner ikke JavaScript: nettleseren laster fortsatt ned hver komponent den skal hydrere, og til hydreringen er ferdig, kan en knapp være tegnet på skjermen uten å gjøre noe.

Det gjør det til et valg per side og ikke per prosjekt. En side som er lik for alle lesere, vinner lite på å bli rendret på nytt ved hver forespørsel, mens en side som endrer seg med leseren, eller med data som flytter seg fra minutt til minutt, er der SSR gjør seg fortjent til serveren sin. Dette nettstedet ligger i den statiske enden. Teknologivalgene er statisk først der det passer, slik at disse sidene ikke krever kjøring av applikasjonskode ved hver forespørsel, og Hugo bygger nettstedet.

Slik ser det ut i praksis

Siden nedenfor er skrevet for denne artikkelen og bruker Pages Router. Den viser ordrene til en innlogget kunde, sender en besøkende uten økt videre til innloggingssiden og ber alle cacher om ikke å ta vare på resultatet. `readSession` og `db` står for det applikasjonen bruker til økter og 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 kjører funksjonen ved hver forespørsel til `/orders`. `req.cookies` er Cookie-headeren i forespørselen, ferdig parset. En `redirect` med `permanent: false` er en midlertidig omdirigering og ikke en permanent. Det som kommer tilbake under `props`, må tåle `JSON.stringify`, og det når også nettleseren som en del av den første HTML-en, slik at React kan hydrere listen med de samme dataene som serveren brukte. Derfor skjærer funksjonen hver rad ned til de tre feltene siden viser, i stedet for å sende databaseradene rett videre. Når en leser senere kommer via en `next/link`-lenke, ber Next.js serveren om å kjøre funksjonen igjen og får props tilbake som JSON, uten en full sideinnlasting.

Next.js 16-frontenden fra grunnleggerens nåværende arbeid hos NextMariner viser problemet dette løser. Porteføljesiden om frontenden tar utgangspunkt i et innlogget dashbord som skulle føles raskt og samtidig være interaktivt. Henter man alt i nettleseren, blir den første innlastingen treg, og N+1-mønsteret følger med: hver komponent våkner og sender sin egen forespørsel, så én enkelt side blir til en kaskade av rundturer. På dashbordet og i de store katalogoversiktene henter en fabrikk ved navn `createPrefetchedServerPage` nå sidens data på serveren og sender de ferdighentede dataene til klienten, slik at komponentene starter med dataene sine i stedet for at hver av dem må spørre etter dem. Sidene som er flyttet over på den, lastes med én rundtur i stedet for en storm av dem. Prosjektet holder fast på de to andre formene der de passer: offentlige sider genereres statisk fordi de ikke endrer seg fra bruker til bruker, og de virkelig interaktive delene forblir klientrendret. Det finnes også en dokumentert strategi for cache-invalidering for hver dataspørring, slik at dataene holder seg ferske uten at appen henter det den allerede har, en gang til.

Hvor det går galt

Serveren og nettleseren er uenige. Hydrering forventer at nettleserens første rendering gir det samme resultatet som serverens. En komponent som sjekker `typeof window`, kaller `window.matchMedia` under renderingen eller skriver ut klokkeslettet, rendrer én ting på serveren og en annen i nettleseren. Reacts liste over årsaker nevner også ulike data på de to sidene og overflødig mellomrom inne i rotelementet. I beste fall gjør et avvik hydreringen tregere, og i verste fall havner hendelseshåndterere på feil elementer, så Reacts referanse for `hydrateRoot` sier det rett ut: «You should treat mismatches as bugs and fix them.» Rendre det samme på begge sider. Der nettleseren virkelig må vise noe annet, rendrer man serverens versjon først og bytter i en effect etter hydreringen, noe som koster en ekstra rendering. `suppressHydrationWarning` hører hjemme ved en uunngåelig forskjell som et tidsstempel, og den virker bare ett nivå ned.

Hemmeligheter følger med i props. Alt som returneres som props, skrives inn i den første HTML-en slik at nettleseren kan hydrere, og Next.js-dokumentasjonen advarer mot å sende noe sensitivt den veien. En funksjon som returnerer en hel brukerpost, gir passordhashen og de interne flaggene til alle som åpner kildekoden til siden. Returner bare feltene siden viser, slik eksempelet over gjør.

En delt cache gir én lesers side til den neste. Next.js-dokumentasjonen viser caching av serverrenderte sider med `Cache-Control: public, s-maxage=10, stale-while-revalidate=59` . Kopiert over på en personlig side inviterer den headeren et CDN eller en proxy foran serveren til å ta vare på svaret og gi det til den som spør etterpå. RFC 9111 definerer `public` som tillatelse for en cache til å lagre et svar den ellers ville avvist, og `s-maxage` som en levetid beregnet på delte cacher. Hver av dem opphever dessuten regelen om at en delt cache ikke kan gjenbruke svaret på en forespørsel som hadde med en `Authorization` -header. En side bygd av informasjonskapsler eller innloggingsdata trenger `private` , som forbyr delte cacher å lagre den, eller `no-store` , som forbyr alle cacher det.

Det tregeste oppslaget holder siden tilbake. Rendres siden i ett stykke, sender den ingenting før hvert eneste oppslag bak den er ferdig, så ett tregt oppslag forsinker toppen av siden, navigasjonen og alt annet. Reacts strømmende renderer deler siden ved `<Suspense>` -grenser. Den sender skallet, alt utenfor grensene, så snart det er klart, med fallbacks på plass, og strømmer deretter hver gjenværende del med et lite inline-skript som setter den inn. Det følger én hake med å sende tidlig: når strømmingen har begynt, kan statuskoden ikke lenger endres, så en manglende post eller en avvist sjekk må være avgjort før skallet sendes, og `onShellError` er stedet der en feilstatus settes.

Les mer

- Server-side Rendering (SSR) — Next.js om HTML laget ved hver forespørsel, og hvordan `getServerSideProps` skiller seg fra `getStaticProps` .
- getServerSideProps — når den skal brukes, hva som når klienten i den første HTML-en, og caching med `Cache-Control` .
- Server React DOM APIs — de strømmende API-ene og legacy-API-ene React tilbyr for å rendre komponenter til HTML på serveren.
- hydrateRoot — hvordan React tar over serverrendret HTML, og de vanlige årsakene til avvik under hydrering.
- renderToPipeableStream — skallet, `<Suspense>` -grenser og forbeholdet om statuskoder ved strømming.
- RFC 9111: HTTP Caching — direktivene `public` , `private` , `s-maxage` og `no-store` og reglene for delte cacher.

Ordliste

Velg ut fra deres faktiske rammer

Beskriv arbeidsflyten, eksisterende grensesnitt og brukerne som trenger endringen. Vi kan definere en testbar leveranse og en overlevering som gjør teamet i stand til å vedlikeholde løsningen.

Se hva det koster Ta kontakt →

Gjør begrepene praktiske

Spør hvordan «Hva er server-side rendering?» gjelder i organisasjonen deres. Et konkret eksempel fra dagens system er nok til å starte samtalen.

Se tjenestene våre

welcome@engineer.company +45 50 10 61 84 København, Danmark

Ta med deg denne siden

Visittkort

Laget med kjærlighet av D.

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

Opphavsrett © 2025 – nå · Engineer ApS · Alle rettigheter forbeholdt.

Skrevet ut fra https://engineer.company/nb/glossary/server-side-rendering/