Engineer

Server-side rendering

att skapa en sidas HTML på servern vid varje begäran

Vad det är

Server-side rendering, oftast förkortat SSR och ibland kallat serverrendering, betyder att en sidas HTML skapas på servern i samma ögonblick som någon ber om den. Webbläsaren får ett färdigt dokument med innehållet på plats och kan visa det innan något skript har körts. I ett JavaScript-ramverk körs samma komponenter sedan en gång till i webbläsaren, som tar över den HTML den fick och gör den interaktiv.

Begreppet svarar på en fråga om tidpunkt: när, och var, skapas HTML-koden? Tre svar är vanliga. En statisk webbplatsgenerator skapar den en gång, när webbplatsen byggs, och skickar samma fil till alla. Server-side rendering skapar den på servern vid varje förfrågan, så att sidan kan bero på vem som frågar och på vilka data som gäller just då. Klientside-rendering, ofta kallad CSR, skickar ett nästan tomt dokument och ett paket JavaScript som bygger sidan i webbläsaren. Verkliga applikationer blandar ofta alla tre, en route i taget.

Next.js säger samma sak. Dokumentationen anger dynamic rendering som ett annat namn på SSR och säger att sidans HTML genereras vid varje förfrågan. I Pages Router väljer en sida det genom att exportera en asynkron funktion som heter `getServerSideProps` och som servern anropar vid varje förfrågan. Syskonfunktionen `getStaticProps` gör samma jobb en gång, när webbplatsen byggs. React står för maskineriet under ytan. API:erna i `react-dom/server` gör om ett träd av komponenter till HTML på servern, antingen som en ström, med `renderToPipeableStream` för Node.js och `renderToReadableStream` för Web Streams, eller i ett stycke med den äldre `renderToString`, som React listar som ett legacy-API med begränsad funktionalitet.

Den andra halvan sker i webbläsaren och kallas hydrering. Next.js beskriver den första inläsningen så här: HTML-koden ger en snabb, icke-interaktiv förhandsvisning, och sedan hydrerar JavaScript komponenterna. Hydrering definierar Next.js som Reacts sätt att koppla händelsehanterare till DOM:en, så att den statiska HTML-koden blir interaktiv. Reacts referens för `hydrateRoot` sticker inte under stol med avvägningen. Ögonblicksbilden i HTML-koden får appen att se ut att laddas snabbare, och sidan ser färdig ut en stund innan den är det.

Använd begreppet i ett beslut

Använd ”Vad är server-side rendering?” i ett konkret beslut. Vi kan knyta begreppet till systemet ni bygger eller köper. Gränssnitt som laddar snabbt, fungerar i mobilen och beter sig likadant i alla webbläsare dina kunder råkar använda.

Se våra tjänster Hör av er →

Granska antagandena bakom ordet

Använd förklaringen och det länkade arbetet för att formulera frågor till projektet. Vi kan undersöka antagandena tillsammans innan ett verktyg väljs. Styrde full stack-utvecklingen av GIS-kartor — kartutveckling i full stack, från databas till webbläsare

Styrde full stack-utvecklingen av GIS-kartor Hör av er →

Varför det spelar roll

Innehållet finns med i det första svaret. En läsare på en långsam uppkoppling ser text och bilder medan JavaScript fortfarande är på väg, och en sökrobot som inte kör JavaScript ser samma sida som en människa och inte ett tomt skal. Webbläsaren behöver inte först köra ett skript och sedan hämta data innan den kan rita upp riktigt innehåll.

Sidan kan också bero på förfrågan, och det är just det som skiljer den från ett statiskt bygge. Next.js rekommenderar `getServerSideProps` för data som är personliga för användaren eller blir kända först vid förfrågan, till exempel en authorization-header eller besökarens plats, och hänvisar allt annat till generering när webbplatsen byggs. Data hämtas nära den plats där de finns. Koden i funktionen körs bara på servern, dess importer hålls utanför klientens bundle, och den kan fråga en databas eller ett CMS direkt. En tur till servern kan ersätta de förfrågningar som en webbläsare annars skulle skicka efter varandra när skripten väl hade laddats.

Kostnaderna är spegelbilden av det en statisk webbplats sparar. Det måste köras en server, och den gör renderingsarbetet på nytt vid varje förfrågan. Svaret kan börja först när data finns på plats, så utan streaming avgör den långsammaste datafrågan tempot för hela sidan. Och rendering på servern tar inte bort

JavaScript: webbläsaren laddar fortfarande ner varje komponent den ska hydrera, och tills hydreringen är klar kan en knapp vara ritad på skärmen utan att göra något.

Det gör det till ett val per sida och inte per projekt. En sida som är likadan för alla läsare vinner lite på att renderas på nytt vid varje förfrågan, medan en sida som ändras med läsaren, eller med data som rör sig från minut till minut, är där SSR gör sig förtjänt av sin server. Den här webbplatsen ligger i den statiska änden. Dess teknikstack är statisk först där det passar, så att de här sidorna inte kräver att applikationskod körs vid varje förfrågan, och Hugo bygger webbplatsen.

Så ser det ut i praktiken

Sidan nedan är skriven för den här artikeln och använder Pages Router. Den visar en inloggad kunds ordrar, skickar en besökare utan session vidare till inloggningssidan och ber alla cachar att inte spara resultatet. `readSession` och `db` står för det applikationen använder för sessioner och 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 kör funktionen vid varje förfrågan till `/orders`. `req.cookies` är förfrågans Cookie-header, redan tolkad. En `redirect` med `permanent: false` är en tillfällig omdirigering och inte en permanent. Det som kommer tillbaka under `props` måste klara `JSON.stringify`, och det når också webbläsaren som en del av den första HTML-koden, så att React kan hydrera listan med samma data som servern använde. Därför skär funktionen ner varje rad till de tre fält som sidan visar, i stället för att skicka databasraderna rakt igenom. När en läsare senare kommer via en `next/link`-länk ber Next.js servern att köra funktionen igen och får tillbaka props som JSON, utan en fullständig sidladdning.

Frontenden i Next.js 16 från grundarens pågående arbete hos NextMariner visar problemet som det här löser. Portfoliosidan om frontenden utgår från en dashboard för inloggade användare som skulle kännas snabb och samtidigt vara interaktiv. Hämtar man allt i webbläsaren blir den första inläsningen seg, och N+1-mönstret följer med: varje komponent vaknar och skickar sin egen förfrågan, så att en enda sida blir en kaskad av rundresor. På dashboarden och de stora katalogsidorna hämtar en fabrik som heter `createPrefetchedServerPage` nu sidans data på servern och lämnar över färdighämtade data till klienten, så att komponenterna börjar med sina data i stället för att var och en ska fråga efter dem. Sidor som har flyttats över till fabriken laddas med en enda rundresa i stället för en storm av dem. Projektet håller fast vid de två andra formerna där de passar: publika sidor genereras statiskt eftersom de inte ändras från användare till användare, och de genuint interaktiva delarna förblir klientrenderade. Det finns också en dokumenterad strategi för cacheinvalidering per datahämtning, så att data håller sig färska utan att appen hämtar det den redan har en gång till.

Var det går fel

Servern och webbläsaren är oense. Hydrering förväntar sig att webbläsarens första rendering ger samma resultat som serverns. En komponent som kontrollerar `typeof window`, anropar `window.matchMedia` under renderingen eller skriver ut aktuell tid renderar en sak på servern och en annan i webbläsaren. Reacts lista över orsaker tar också upp olika data på de två sidorna och överflödiga blanksteg inuti rotelementet. I bästa fall gör en avvikelse hydreringen långsammare, och i värsta fall hamnar händelsehanterare på fel element, så Reacts referens för `hydrateRoot` säger det rakt ut: ”You should treat

mismatches as bugs and fix them.” Rendera samma sak på båda sidor. Där webbläsaren verkligen måste visa något annat renderar man serverns version först och byter i en effect efter hydreringen, vilket kostar en extra rendering. `suppressHydrationWarning` hör hemma vid en oundviklig skillnad som en tidsstämpel, och den verkar bara en nivå ner.

Hemligheter följer med i props. Allt som returneras som props skrivs in i den första HTML-koden så att webbläsaren kan hydrera, och Next.js-dokumentationen varnar för att skicka något känsligt den vägen. En funktion som returnerar en hel användarpost lämnar ut dess lösenordshash och interna flaggor till alla som öppnar sidans källkod. Returnera bara de fält som sidan visar, så som exemplet ovan gör.

En delad cache ger en läsares sida till nästa. Next.js-dokumentationen visar cachning av serverrenderade sidor med `Cache-Control: public, s-maxage=10, stale-while-revalidate=59` . Kopierad till en personlig sida bjuder den headern in ett CDN eller en proxy framför servern att spara svaret och lämna det till den som frågar efteråt. RFC 9111 definierar `public` som tillåtelse för en cache att lagra ett svar som den annars skulle vägra, och `s-maxage` som en livstid avsedd för delade cachar. Var och en av dem upphäver dessutom regeln att en delad cache inte får återanvända svaret på en förfrågan som hade en `Authorization` -header. En sida byggd av kakor eller inloggningsuppgifter behöver `private` , som förbjuder delade cachar att lagra den, eller `no-store` , som förbjuder alla cachar.

Den långsammaste frågan håller tillbaka sidan. Renderas sidan i ett stycke skickar den ingenting förrän varje fråga bakom den är klar, så en enda långsam fråga försenar sidhuvudet, navigeringen och allt annat. Reacts renderare för streaming delar sidan vid `<Suspense>` -gränser. Den skickar skalet, allt utanför gränserna, så snart det är klart, med fallbacks på plats, och streamar sedan varje återstående del med ett litet inline-skript som sätter in den. Det finns en hake med att skicka tidigt: när streamingen har börjat kan statuskoden inte längre ändras, så en saknad post eller en kontroll som inte går igenom måste vara avgjord innan skalet skickas, och `onShellError` är stället där en felstatus sätts.

Läs mer

- Server-side Rendering (SSR) — Next.js om HTML som skapas vid varje förfrågan, och hur `getServerSideProps` skiljer sig från `getStaticProps` .
- getServerSideProps — när den ska användas, vad som når klienten i den första HTML-koden, och cachning med `Cache-Control` .

- Server React DOM APIs — de streaming- och legacy-API:er som React erbjuder för att rendera komponenter till HTML på servern.
- hydrateRoot — hur React tar över serverrenderad HTML, och de vanliga orsakerna till avvikelser vid hydrering.
- renderToPipeableStream — skalet, `<Suspense>` -gränser och förbehållet om statuskoder vid streaming.
- RFC 9111: HTTP Caching — direktiven `public` , `private` , `s-maxage` och `no-store` och reglerna för delade cachar.

Ordlista

Välj utifrån era faktiska förutsättningar

Beskriv arbetsflödet, befintliga gränssnitt och användarna som behöver förändringen. Vi kan definiera en testbar leverans och en överlämning som gör att teamet kan underhålla lösningen.

Se vad det kostar Hör av er →

Gör begreppen praktiska

Fråga hur ”Vad är server-side rendering?” gäller i er organisation. Ett konkret exempel från ert nuvarande system räcker för att börja samtalet.

Se våra tjänster

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

Ta med dig den här sidan

Visitkort

Omsorgsfullt skapad med kärlek av D.

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

Upphovsrätt © 2025 – nu · Engineer ApS · Alla rättigheter förbehållna.

Utskrivet från https://engineer.company/sv/glossary/server-side-rendering/