Engineer

Rendu côté serveur

HTML d'une page produit sur le serveur à chaque requête

Ce que c'est

Le rendu côté serveur, généralement abrégé en SSR, consiste à produire le HTML d'une page sur le serveur au moment où quelqu'un le demande. Le navigateur reçoit un document complet dont le contenu est déjà en place et peut l'afficher avant l'exécution du moindre script. Dans un framework JavaScript, les mêmes composants s'exécutent ensuite une seconde fois dans le navigateur, qui reprend le HTML reçu et le rend interactif.

Le terme répond à une question de moment et de lieu : quand et où le HTML est-il produit ? Trois réponses sont courantes. Un générateur de site statique le produit une fois, à la construction, et sert le même fichier à tout le monde. Le rendu côté serveur le produit sur le serveur à chaque demande : la page peut ainsi dépendre de la personne qui la demande et des données à cet instant. Le rendu côté client, souvent abrégé en CSR, envoie un document presque vide et un paquet de code JavaScript qui construit la page dans le navigateur. Les applications réelles combinent souvent les trois, route par route.

Next.js le présente de la même façon. Sa documentation donne le rendu dynamique comme autre nom du SSR et précise que le HTML de la page est généré à chaque demande. Dans le Pages Router, une page l'active en exportant une fonction asynchrone appelée `getServerSideProps`, que le serveur appelle à chaque demande ; sa fonction sœur, `getStaticProps`, remplit le même rôle une seule fois, à la construction. React fournit les mécanismes sous-jacents. Ses API `react-dom/server` transforment un arbre de composants en HTML sur le serveur : soit en flux, avec `renderToPipeableStream` pour Node.js et `renderToReadableStream` pour les Web Streams, soit en un seul bloc avec l'ancienne `renderToString`, que React classe parmi les API historiques aux fonctionnalités limitées.

La seconde moitié se déroule dans le navigateur et s'appelle l'hydratation. Next.js décrit ainsi le premier chargement : le HTML fournit rapidement un aperçu non interactif, puis JavaScript hydrate les composants. Il définit cette étape comme le raccordement, par React, des gestionnaires d'événements au DOM existant pour que la page réagisse aux actions de l'utilisateur. La documentation de React pour `hydrateRoot` expose clairement le compromis. L'instantané HTML donne l'impression que l'application se charge plus vite, et la page paraît prête un instant avant de l'être réellement.

Utilisez le concept pour prendre une décision

Utilisez « Qu'est-ce que le rendu côté serveur ? » pour éclairer une décision réelle. Nous pouvons vous aider à relier ce concept au système que vous construisez ou achetez. Des interfaces rapides, utilisables sur téléphone et cohérentes dans tous les navigateurs de vos clients.

Découvrir nos prestations Contactez-nous →

Vérifiez les hypothèses derrière le terme

Appuyez-vous sur l'explication et les réalisations liées pour formuler les questions de votre projet. Nous pouvons examiner les hypothèses avec vous avant le choix d'un outil. Développement cartographique GIS full-stack — cartographie full-stack, de la base au navigateur

Développement cartographique GIS full-stack Contactez-nous →

Pourquoi c'est important

Le contenu se trouve dans la première réponse. Sur une connexion lente, le lecteur voit du texte et des images pendant que JavaScript se télécharge encore. Un robot qui n'exécute pas JavaScript voit la même page qu'une personne, plutôt qu'une coquille vide. Le navigateur n'a pas à exécuter un script, puis à récupérer des données, avant de pouvoir afficher du véritable contenu.

La page peut aussi dépendre de la demande, ce qui la distingue d'une construction statique. Next.js recommande `getServerSideProps` pour les données propres à l'utilisateur ou connues seulement au moment de la demande, comme un en-tête d'autorisation ou la localisation du visiteur, et oriente tout le reste vers la génération à la construction. Les données sont récupérées près de l'endroit où elles résident. Le code de cette fonction s'exécute uniquement sur le serveur, ses imports sont exclus du bundle client, et il peut interroger directement une base de données ou un système de gestion de contenu. Un seul aller-retour avec le serveur peut remplacer les demandes qu'un navigateur enchaînerait autrement après le chargement de ses scripts.

Les coûts sont le reflet des économies d'un site statique. Il faut un serveur en fonctionnement, qui recommence le rendu à chaque demande. La réponse ne peut commencer que lorsque les données sont disponibles : sans diffusion en

flux, la requête la plus lente impose donc son rythme à toute la page. Le rendu côté serveur ne supprime pas non plus JavaScript. Le navigateur télécharge encore chaque composant qu'il doit hydrater ; tant que l'hydratation n'est pas terminée, un bouton peut être dessiné à l'écran sans rien faire.

Le choix se fait donc page par page, plutôt que pour tout un projet. Une page identique pour chaque lecteur gagne peu à être recalculée à chaque requête. Le SSR justifie son serveur lorsque la page change selon le lecteur ou selon des données qui évoluent à la minute. Ce site se situe à l'extrémité statique. Sa pile technique privilégie le statique là où il convient, pour que ces pages ne nécessitent pas l'exécution d'une application au moment de la requête ; Hugo construit le site.

À quoi cela ressemble en pratique

La page ci-dessous, écrite pour cet article, utilise le Pages Router. Elle liste les commandes d'un client connecté, redirige un visiteur sans session vers la page de connexion et demande à tous les caches de ne pas conserver le résultat. `readSession` et `db` représentent les mécanismes de session et de données que l'application utilise.

```
// 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 exécute la fonction à chaque requête vers `/orders`. `req.cookies` est l'en-tête de cookies de la requête, déjà analysé. Une valeur `redirect` avec `permanent: false` produit une redirection temporaire plutôt que permanente. Tout ce qui revient sous `props` doit pouvoir être sérialisé avec `JSON.stringify` et atteint aussi le navigateur, dans le HTML initial, afin que React hydrate la liste avec les mêmes données que le serveur. C'est pourquoi la fonction réduit chaque ligne aux trois champs affichés, au lieu de transmettre directement les lignes de la base. Lorsqu'un lecteur arrive ensuite par un lien `next/link`, Next.js demande au serveur d'exécuter de nouveau la fonction et récupère les props en JSON, sans charger une page complète.

Le frontend Next.js 16, sur lequel la fondatrice travaille actuellement dans le cadre de NextMariner, illustre le problème que cela résout. La réalisation correspondante part d'un tableau de bord authentifié qui devait paraître rapide tout en restant interactif. Tout récupérer dans le navigateur ralentit le premier chargement et introduit le schéma N+1 : chaque composant se réveille et envoie sa propre requête, transformant une page en cascade d'allers-retours. Sur ce tableau de bord et dans les grandes listes d'annuaire, une fabrique appelée `createPrefetchedServerPage` récupère désormais les données de la page sur le serveur, puis les transmet au client déjà chargées. Les composants démarrent donc avec leurs données, au lieu de partir chacun les demander. Les pages migrées vers cette fabrique s'affichent après un seul aller-retour, plutôt qu'après une multitude d'allers-retours. La même réalisation conserve les deux autres modes là où ils conviennent : les pages publiques sont générées statiquement, car elles ne changent pas d'un utilisateur à l'autre, et les parties réellement interactives restent rendues côté client. Elle documente aussi une stratégie d'invalidation du cache pour chaque requête de données, afin de garder les données à jour sans récupérer à nouveau ce que l'application possède déjà.

Où les problèmes surviennent

Le serveur et le navigateur ne sont pas d'accord. L'hydratation attend que le premier rendu du navigateur produise le même résultat que celui du serveur. Un composant qui teste `typeof window`, appelle `window.matchMedia` pendant le rendu ou affiche l'heure courante produit une chose sur le serveur et une autre dans le navigateur. React cite aussi des données différentes des deux côtés et des espaces indésirables à l'intérieur de l'élément racine. Au mieux, une divergence ralentit l'hydratation ; au pire, les gestionnaires d'événements se retrouvent attachés aux mauvais éléments. Sa documentation pour `hydrateRoot` est donc explicite : « Vous devriez considérer les divergences comme des bugs et les corriger. » Produisez le même résultat des deux côtés. Lorsque le navigateur doit réellement afficher autre chose, commencez par la version du serveur, puis changez-la dans un effet après l'hydratation, au prix d'un second rendu. Réservez `suppressHydrationWarning` à une différence inévitable, comme un horodatage, en gardant à l'esprit qu'il ne fonctionne que sur un niveau de profondeur.

Les secrets voyagent dans les props. Tout ce qui est renvoyé comme props est écrit dans le HTML initial pour que le navigateur puisse hydrater la page. La documentation de Next.js déconseille donc d'y transmettre des informations sensibles. Une fonction qui renvoie un enregistrement utilisateur complet expose à quiconque ouvre le code source de la page l'empreinte du mot de passe de l'utilisateur et ses indicateurs internes. Ne renvoyez que les champs affichés, comme dans l'exemple ci-dessus.

Un cache partagé sert la page d'un lecteur au suivant. La documentation de Next.js illustre la mise en cache des pages rendues côté serveur avec `Cache-Control: public, s-maxage=10, stale-while-revalidate=59`. Copié sur une page personnalisée, cet en-tête invite un réseau de diffusion de contenu ou un mandataire placé devant le serveur à garder la réponse et à la donner au prochain demandeur. La RFC 9111 définit `public` comme l'autorisation, pour un cache, de stocker une réponse qu'il refuserait autrement, et `s-maxage` comme une durée de vie destinée aux caches partagés. Chacune de ces directives lève aussi la règle qui empêche un cache partagé de réutiliser une réponse à une requête portant un en-tête `Authorization`. Une page construite à partir de cookies ou d'informations d'authentification a besoin de `private`, qui interdit son stockage aux caches partagés, ou de `no-store`, qui l'interdit à tous les caches.

La requête la plus lente bloque la page. Rendue en un seul bloc, la page n'envoie rien tant que toutes les requêtes sous-jacentes ne sont pas terminées : une requête lente retarde donc l'en-tête, la navigation et tout le reste. Le moteur de rendu en flux de React découpe la page aux frontières `<Suspense>`. Il envoie la structure initiale, c'est-à-dire tout ce qui se trouve hors de ces frontières, dès qu'elle est prête, avec des contenus de remplacement, puis diffuse chaque partie restante avec un petit script inclus dans le HTML qui remplace le contenu provisoire par cette partie. L'envoi précoce a une contrepartie : une fois la diffusion commencée, le code de statut ne peut plus changer. Il faut donc trancher le cas d'un enregistrement absent ou d'un contrôle échoué avant d'envoyer cette structure initiale ; `onShellError` est l'endroit où définir un statut d'erreur.

Pour aller plus loin

- Rendu côté serveur (SSR) — Next.js sur le HTML généré à chaque requête et la différence entre `getServerSideProps` et `getStaticProps`.
- getServerSideProps — Quand l'utiliser, ce qui parvient au client dans le HTML initial et la mise en cache avec `Cache-Control`.
- API React DOM pour le serveur — Les API en flux et historiques de React pour produire du HTML à partir de composants sur le serveur.
- hydrateRoot — Comment React reprend le HTML produit sur le serveur et les causes courantes des divergences d'hydratation.
- renderToPipeableStream — La structure initiale, les frontières `<Suspense>` et la limite concernant le statut lors de la diffusion en flux.
- RFC 9111 : mise en cache HTTP — Les directives `public`, `private`, `s-maxage` et `no-store`, et les règles applicables aux caches partagés.

Glossaire

Choisissez selon vos contraintes réelles

Présentez le processus de travail, les interfaces existantes et les utilisateurs qui ont besoin du changement. Nous pouvons définir une livraison testable et une transmission permettant à votre équipe d'assurer la maintenance des éléments livrés.

Consulter les tarifs Contactez-nous →

Rendez la terminologie concrète

Demandez comment « Qu'est-ce que le rendu côté serveur ? » s'applique à votre organisation. Un exemple concret tiré de votre système actuel suffit pour engager la discussion.

Découvrir nos prestations

welcome@engineer.company +45 50 10 61 84 Copenhague, Danemark

Emportez cette page avec vous

Fiche de contact

Créé avec soin et amour par D.

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

Droit d'auteur © 2025 – aujourd'hui · Engineer ApS · Tous droits réservés.

Imprimé depuis https://engineer.company/fr/glossary/server-side-rendering/