

Engineer

renderizado del lado del servidor

Generar el HTML de una página en el servidor por solicitud



Qué es

El renderizado del lado del servidor, normalmente abreviado como SSR, consiste en producir el HTML de una página en el servidor en el momento en que alguien la solicita. El navegador recibe un documento completo, con el contenido ya colocado, y puede mostrarlo antes de que se haya ejecutado ningún script. En un framework de JavaScript, los mismos componentes se ejecutan después una segunda vez en el navegador, que toma el control del HTML recibido y lo hace interactivo.

El término responde a una pregunta sobre el momento y el lugar: ¿cuándo y dónde se crea el HTML? Hay tres respuestas habituales. Un generador de sitios estáticos lo crea una vez, durante la construcción, y sirve el mismo archivo a todo el mundo. El renderizado del lado del servidor lo crea en el servidor para cada solicitud, por lo que la página puede depender de quién la pide y de lo que digan los datos en ese momento. El renderizado del lado del cliente, a menudo llamado CSR, envía un documento casi vacío y un paquete de JavaScript que construye la página en el navegador. Las aplicaciones reales suelen combinar las tres modalidades, ruta por ruta.

Next.js lo explica de la misma manera. Su documentación utiliza renderizado dinámico como otro nombre para SSR y dice que el HTML de la página se genera en cada solicitud. En el Pages Router, una página lo activa exportando una función asíncrona llamada `getServerSideProps`, que el servidor llama en cada solicitud. Su función hermana, `getStaticProps`, hace ese trabajo una sola vez, durante la construcción. React aporta la maquinaria subyacente. Sus API `react-dom/server` convierten un árbol de componentes en HTML en el servidor, ya sea como flujo, con `renderToPipeableStream` para Node.js y `renderToReadableStream` para Web Streams, o en una sola pieza con la antigua `renderToString`, que React clasifica como una API heredada de funcionalidad limitada.

La segunda mitad ocurre en el navegador y se llama hidratación. Next.js describe así la primera carga: el HTML ofrece una vista previa rápida y no interactiva, y después JavaScript hidrata los componentes. Define esa hidratación como la conexión que hace React entre los manejadores de eventos y el DOM existente para que la página responda a las entradas del usuario. La referencia de React sobre `hydrateRoot` expone la contrapartida con franqueza. La instantánea HTML hace que la aplicación parezca cargar antes, y la página parece lista un momento antes de estarlo.

Utilice el concepto en una decisión

Aplique «¿Qué es el renderizado del lado del servidor?» a una decisión real. Podemos ayudarle a relacionar el concepto con el sistema que está construyendo o comprando. Interfaces que cargan rápido, funcionan en el teléfono y se comportan igual en los navegadores que utilizan sus clientes.

Explore nuestros servicios Póngase en contacto →

Compruebe los supuestos detrás del término

Utilice la explicación y los trabajos enlazados para formular preguntas sobre su proyecto. Podemos examinar los supuestos con usted antes de elegir una herramienta. Dirección del desarrollo integral de mapas GIS — cartografía integral, de la base de datos al navegador

Dirección del desarrollo integral de mapas GIS Póngase en contacto →

Por qué importa

El contenido está en la primera respuesta. Una persona con una conexión lenta ve texto e imágenes mientras JavaScript aún está en camino, y un rastreador que no ejecuta JavaScript ve la misma página que una persona, en lugar de una estructura vacía. El navegador no tiene que ejecutar un script y después obtener datos antes de poder mostrar contenido real.

La página también puede depender de la solicitud, lo que la distingue de una construcción estática. Next.js recomienda `getServerSideProps` para datos específicos del usuario o que solo se conocen al recibir la solicitud, como una cabecera de autorización o la ubicación del visitante, y orienta todo lo demás hacia la generación durante la construcción. Los datos se obtienen cerca de donde residen. El código de esa función solo se ejecuta en el servidor, sus importaciones no se incluyen en el paquete del cliente y puede consultar directamente una base de datos o un sistema de gestión de contenidos. Una ida y vuelta al servidor puede sustituir a las solicitudes que, de otro modo, el navegador haría una tras otra después de cargar sus scripts.

Los costes son la contrapartida de los ahorros de un sitio estático. Tiene que haber un servidor en ejecución, que vuelve a renderizar con cada solicitud. La respuesta solo puede empezar cuando están disponibles los datos: sin

transmisión progresiva, la consulta más lenta marca el ritmo de toda la página. Y el renderizado en el servidor no elimina JavaScript. El navegador sigue descargando todos los componentes que debe hidratar y, hasta que termina la hidratación, un botón puede aparecer en pantalla y no hacer nada.

Por eso la elección corresponde a cada página, no a todo el proyecto. Una página idéntica para todos sus lectores gana poco si se vuelve a renderizar en cada solicitud. SSR justifica su servidor cuando la página cambia según el lector o según datos que varían de un minuto a otro. Este sitio está en el extremo estático. Su conjunto de tecnologías prioriza lo estático cuando encaja, para que esas páginas no requieran ejecutar una aplicación al atender una solicitud, y Hugo construye el sitio.

Cómo se aplica en la práctica

La página siguiente, escrita para este artículo, utiliza el Pages Router. Enumera los pedidos de un cliente que ha iniciado sesión, envía a quien no tenga sesión a la página de acceso e indica a todas las cachés que no guarden el resultado. `readSession` y `db` representan los mecanismos que la aplicación utilice para las sesiones y los datos.

```
// 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 ejecuta la función en cada solicitud a `/orders`. `req.cookies` es la cabecera de cookies de la solicitud, ya analizada. Un `redirect` con `permanent: false` es una redirección temporal, no permanente. Todo lo devuelto bajo `props` tiene que poder serializarse con `JSON.stringify` y también llega al navegador como parte del HTML inicial, para que React hidrate la lista con los mismos datos que utilizó el servidor. Por eso la función reduce cada fila a los tres campos que muestra la página, en lugar de devolver directamente las filas de la base de datos. Cuando más adelante una persona llega mediante un enlace `next/link`, Next.js pide al servidor que vuelva a ejecutar la función y recibe las propiedades como JSON, sin cargar de nuevo la página completa.

En el trabajo actual de la fundadora en NextMariner, el frontend en Next.js 16 muestra el problema que esto resuelve. Su entrada del portfolio parte de un panel con autenticación que debía resultar ágil y mantenerse interactivo. Obtenerlo todo en el navegador ralentiza la primera carga e introduce el patrón N+1: cada componente se activa y envía su propia solicitud, de modo que una sola página se convierte en una cascada de idas y vueltas. En ese panel y en los grandes listados del directorio, una factoría llamada `createPrefetchedServerPage` obtiene ahora los datos de la página en el servidor y los entrega al cliente ya cargados. Los componentes arrancan con sus datos, en lugar de salir cada uno a pedirlos. Las páginas trasladadas a la factoría aparecen con una sola ida y vuelta, en lugar de una avalancha de ellas. La misma entrada conserva las otras dos modalidades donde encajan: las páginas públicas se generan estáticamente porque no cambian entre usuarios, y las partes realmente interactivas siguen renderizándose en el cliente. También documenta una estrategia de invalidación de caché para cada solicitud de datos, de modo que los datos se mantengan actualizados sin volver a obtener lo que la aplicación ya tiene.

Dónde surgen los problemas

El servidor y el navegador no coinciden. La hidratación espera que el primer renderizado del navegador produzca el mismo resultado que el del servidor. Un componente que comprueba `typeof window`, llama a `window.matchMedia` durante

el renderizado o imprime la hora actual produce una cosa en el servidor y otra en el navegador. La lista de causas de React también incluye datos distintos en ambos lados y espacios en blanco sobrantes dentro del elemento raíz. En el mejor de los casos, la discrepancia ralentiza la hidratación; en el peor, los manejadores de eventos quedan conectados a elementos equivocados. Por eso, la referencia de `hydrateRoot` dice con claridad: «Debe tratar las discrepancias como errores y corregirlas». Renderice lo mismo en ambos lados. Cuando el navegador deba mostrar realmente algo distinto, renderice primero la versión del servidor y cámbiela en un efecto después de la hidratación, lo que cuesta un segundo renderizado. Reserve `suppressHydrationWarning` para diferencias inevitables, como una marca de tiempo, y recuerde que solo funciona a un nivel de profundidad.

Los secretos viajan en las propiedades. Todo lo que se devuelve como propiedades se escribe en el HTML inicial para que el navegador pueda hidratar, y la documentación de Next.js advierte de que no se debe enviar así información sensible. Una función que devuelve un registro de usuario completo entrega su hash de contraseña y sus indicadores internos a cualquiera que abra el código fuente de la página. Devuelva únicamente los campos que muestra la página, como hace el ejemplo anterior.

Una caché compartida sirve la página de un lector al siguiente. La documentación de Next.js ilustra el almacenamiento en caché de páginas renderizadas en el servidor con `Cache-Control: public, s-maxage=10, stale-while-revalidate=59` . Copiada en una página personalizada, esa cabecera permite que una red de distribución de contenido o un proxy situado delante del servidor conserve la respuesta y la entregue a quien la pida después. La RFC 9111 define `public` como permiso para que una caché guarde una respuesta que de otro modo rechazaría, y `s-maxage` como un tiempo de vigencia destinado a cachés compartidas. Cualquiera de las dos directivas también levanta la prohibición de que una caché compartida reutilice una respuesta a una solicitud que incluía una cabecera `Authorization` . Una página construida a partir de cookies o credenciales necesita `private` , que prohíbe a las cachés compartidas guardarla, o `no-store` , que lo prohíbe a todas las cachés.

La consulta más lenta retiene la página. Si se renderiza en una sola pieza, la página no envía nada hasta que han terminado todas las consultas que necesita. Una sola consulta lenta retrasa así la cabecera, la navegación y todo lo demás. El renderizador progresivo de React divide la página por los límites de `<Suspense>` . Envía la estructura base, todo lo que queda fuera de cualquier límite, en cuanto está lista y con contenido provisional en su lugar. Después transmite cada parte

restante con un pequeño script en línea que sustituye el contenido provisional por ella. Enviar pronto tiene una consecuencia: una vez iniciada la transmisión, ya no puede cambiar el código de estado. Por tanto, un registro inexistente o una comprobación fallida deben resolverse antes de enviar la estructura base, y `onShellError` es donde se establece un estado de error.

Lecturas adicionales

- Renderizado del lado del servidor (SSR) — Next.js sobre el HTML generado en cada solicitud y la diferencia entre `getServerSideProps` y `getStaticProps`.
- getServerSideProps — cuándo utilizarlo, qué llega al cliente en el HTML inicial y el almacenamiento en caché con `Cache-Control`.
- API de React DOM para servidores — las API de transmisión progresiva y heredadas que ofrece React para renderizar componentes como HTML en el servidor.
- hydrateRoot — cómo React toma el control del HTML renderizado en el servidor y las causas habituales de discrepancias de hidratación.
- renderToPipeableStream — la estructura base, los límites de `<Suspense>` y la limitación del código de estado al transmitir progresivamente.
- RFC 9111: almacenamiento en caché HTTP — las directivas `public`, `private`, `s-maxage` y `no-store`, y las reglas para cachés compartidas.

Glosario

Elija según sus restricciones reales

Describa el flujo de trabajo, las interfaces existentes y los usuarios que necesitan el cambio. Podemos definir una entrega comprobable y un traspaso que permita a su equipo mantener lo entregado.

Vea cuánto cuesta Póngase en contacto →

Haga práctica la terminología

Pregunte cómo se aplica «¿Qué es el renderizado del lado del servidor?» a su organización. Basta un ejemplo concreto de su sistema actual para iniciar la conversación.

Explore nuestros servicios

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

Llévese esta página

Tarjeta de contacto

Creado con esmero y cariño por D.

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

Derechos de autor © 2025 – actualidad · Engineer ApS · Todos los derechos reservados.

Impreso desde https://engineer.company/es/glossary/server-side-rendering/