Componentes integrados de Next.js

Next.js proporciona un conjunto de componentes personalizados. Cada uno de estos aborda un caso de uso específico: por ejemplo, acceder a propiedades internas de la página, como el título de la página o metadatos SEO (next/head), mejorar el rendimiento general de la aplicación y la experiencia del usuario (next/image), o habilitar el enrutamiento de la aplicación (next/link).

El componente next/head

El componente next/head exporta un componente Head personalizado específico de Next.js. Lo usamos para configurar el título HTML de una página y elementos meta, que se encuentran dentro de un componente head HTML. Para mejorar el ranking SEO y mejorar la usabilidad, cada página debe tener sus propios metadatos. El siguiente listado muestra un ejemplo de la página hello.tsx con un título personalizado y un elemento meta.

Es importante recordar que los elementos Head no se combinan entre las páginas. El enrutamiento del lado del cliente de Next.js elimina el contenido del elemento Head durante la transición de la página.

```
import type {NextPage} from "next";

import Head from "next/head";

const Hello: NextPage = () => {

   return (

      <div>

         <Head>

            <title>Titulo de la pagina Hola Mundo</title>

            <meta property="og:title" content="Hello World" key="title" />

         </Head>

         <div>Hello World!</div>

      </div>

   );

};

export default Hello;
```

Importamos el elemento Head del componente next/head y lo añadimos al elemento JSX devuelto, colocándolo por encima del contenido existente y envolviendo ambos en otro elemento div porque necesitamos devolver un elemento en lugar de dos.

El componente next/link

El componente next/link exporta un componente Link. Este componente está construido sobre el elemento Link de React. Lo usamos en lugar de una etiqueta de anclaje HTML cuando queremos vincular a otra página en la aplicación, lo que permite transiciones del lado del cliente entre páginas. Al hacer clic, el componente Link actualiza el DOM del navegador con el nuevo DOM, desplaza hacia la parte superior de la nueva página y ajusta el historial del navegador.

Además, proporciona optimizaciones de rendimiento integradas, precargando la página vinculada y sus datos tan pronto como el componente Link entra en la ventana gráfica (la parte visible del sitio web). Este precarga en segundo plano permite transiciones de página suaves. El siguiente listado agrega el elemento Link de Next.js a la página del listado anterior.

```
import type {NextPage} from "next";

import Head from "next/head";

import Link from "next/link";

const Hello: NextPage = () => {

  return (

    <div>

      <Head>

        <title>Titulo de la pagina Hola Mundo </title>

        <meta property="og:title" content="Hello World" key="title" />

      </Head>

      <div>Hello World!</div>

      <div>

        Usa el ancla de HTML para

        <a href="https://nostarch.com"> enlace externo</a>

        Y un componente Link para una

        <Link href="/components/weather"> pagina internal</Link>.

      </div>

    </div>

  );

};

export default Hello;
```

Importamos el componente, luego lo añadimos al elemento JSX devuelto. Para fines de comparación, usamos un ancla HTML regular para vincular a la página de inicio de No Starch Press y

el Link personalizado para conectar a la página del componente weather en nuestra aplicación Next.js. En la aplicación, intenta hacer clic en ambos enlaces para ver la diferencia.

El componente next/image

El componente next/image exporta un componente Image utilizado para mostrar imágenes. Este componente está construido sobre el elemento HTML <img>. Maneja requisitos comunes de diseño, como llenar todo el espacio disponible y escalar imágenes. El componente puede cargar formatos de imagen modernos, como AVIF y WebP, y servir la imagen con el tamaño correcto para la pantalla del cliente.

Además, tienes la opción de usar imágenes de marcador de posición desenfocadas y cargar la imagen real de forma diferida tan pronto como entra en la ventana gráfica; esto refuerza la estabilidad visual de tu sitio web al prevenir cambios de diseño acumulativos, que ocurren cuando una imagen se renderiza después de la página, causando que el contenido de la página se desplace hacia abajo. Los cambios de diseño acumulativos se consideran una mala experiencia de usuario, y pueden hacer que el usuario pierda su enfoque.

El siguiente listado proporciona un ejemplo básico del componente next/image.

```
import type {NextPage} from "next";

import Head from "next/head";

import Link from "next/link";

import Image from "next/image";

const Hello: NextPage = () => {

  return (

    <div>

      <Head>

        <title>Titulo de pagina Hola Mundo</title>

        <meta property="og:title" content="Hello World" key="title" />

      </Head>

      <div>Hello World!</div>

      <div>

        Usa el ancla de HTML para un <a href="https://nostarch.com">enlace externo</a> y el componente Link para una

        <Link href="/components/weather"> pagina interna</Link>.

        <Image src="/vercel.svg"

          alt="Vercel Logo"
```

```
            width={72}
            height={16}
          />
        </div>
      </div>
  );
};
export default Hello;
```

Aquí mostramos el logotipo de Vercel desde la carpeta public de nuestra aplicación. Primero importamos el componente del paquete next/image. Luego lo añadimos al contenido de la página. La sintaxis y las propiedades de nuestro ejemplo mínimo son similares al elemento img de HTML.

Puedes leer más sobre las propiedades avanzadas del componente en la documentación oficial en https://nextjs.org/docs/api-reference/next/image.

Pre-renderizado y publicación

Next.js proporciona tres opciones para pre-renderizar tu aplicación con su servidor incorporado. La primera, generación de sitios estáticos (SSG), genera el HTML en el momento de la construcción. Por lo tanto, cada solicitud siempre devolverá el mismo HTML, que permanece estático y nunca se vuelve a crear. La segunda opción, renderizado del lado del servidor (SSR), genera nuevos archivos HTML en cada solicitud, y la tercera, regeneración estática incremental (ISR), combina ambos enfoques.

Next.js nos permite elegir nuestra opción de pre-renderizado en función de cada página, lo que significa que la aplicación full-stack puede contener páginas con SSG, SSR e ISR, así como renderizado del lado del cliente para algunos componentes React. También puedes crear una exportación estática completa de tu sitio ejecutando next export. La aplicación exportada se ejecutará de forma independiente en todas las infraestructuras, ya que no necesita el servidor incorporado de Next.js.

Para ganar experiencia con estos enfoques de renderizado, crearemos una nueva página que muestre los datos de nuestra API de nombres para cada opción de renderizado. Crea una nueva carpeta, utils, junto a la carpeta pages y añade un archivo vacío, fetch-names.ts, a ella. Luego agrega el siguiente código. Esta función de utilidad llama a la API remota y devuelve el conjunto de datos.

```
type responseItemType = {
  id: string;
  name: string;
};
export const fetchNames = async () => {
```

```
  const url = "https://www.usemodernfullstack.dev/api/v1/users";
  let data: responseItemType[] | [] = [];
  let names: responseItemType[] | [];
  try {
    const response = await fetch(url);
    data = (await response.json()) as responseItemType[];
  } catch (err) {
    names = [];
  }
  names = data.map((item) => {return {id: item.id, name: item.name}});
  return names;
};
```

Después de definir un tipo personalizado, creamos una función y la exportamos directamente. Esta función contiene el código del archivo names.ts creado anteriormente, con dos ajustes: primero necesitamos definir el array data como posiblemente vacío; luego, devolvemos un array vacío en lugar de una cadena de error si la llamada a la API falla. Este cambio significa que no necesitamos verificar el tipo antes de iterar sobre el array cuando generamos la cadena JSX.

Renderizado del lado del servidor

Usando SSR, el servidor Node.js incorporado de Next.js crea el HTML de una aplicación en respuesta a cada solicitud. Deberías usar esta técnica si tu página depende de datos frescos de una API externa. Desafortunadamente, SSR es más lento en producción, porque las páginas no son fácilmente almacenables en caché.

Para usar SSR para una página, exporta una función async adicional, getServerSideProps, desde esa página. Next.js llama a esta función en cada solicitud y pasa los datos obtenidos al argumento props de la página para pre-renderizarla antes de enviarla al cliente.

Prueba esto creando un nuevo archivo, names-ssr.tsx, en la carpeta pages. Pega el siguiente código en el archivo:

```
import type {
  GetServerSideProps,
  GetServerSidePropsContext,
  InferGetServerSidePropsType,
  NextPage,
  PreviewData
```

```
} from "next";
import {ParsedUrlQuery} from "querystring";
import {fetchNames} from "../utils/fetch-names";

type responseItemType = {
   id: string;
   name: string;
};
const NamesSSR: NextPage = (props: InferGetServerSidePropsType<typeof getServerSideProps>) => {
   const output = props.names.map((item: responseItemType, idx: number) => {
      return (
         <li key={name-${idx}}>
            {item.id} : {item.name}
         </li>
      );
   });
   return (
      <ul>
         {output}
      </ul>
   );
};
export const getServerSideProps: GetServerSideProps = async (
   context: GetServerSidePropsContext<ParsedUrlQuery, PreviewData>
) => {
   let names: responseItemType[] | [] = [];
   try {
      names = await fetchNames();
   } catch(err) {}
```

```
  return {
    props: {
      names
    }
  };
};

export default NamesSSR;
```

Para usar el SSR de Next.js, exportamos la función async adicional, getServerSideProps. También importamos la funcionalidad necesaria de los paquetes next y querystring, y la función fetchNames que creamos anteriormente. Luego, definimos el tipo personalizado para la respuesta a la solicitud de la API.

A continuación, creamos la página y almacenamos la exportación como la predeterminada. La página devuelve un NextPage y toma las propiedades predeterminadas para este tipo de página. Iteramos sobre el array names del parámetro props y creamos una cadena JSX que renderizamos y devolvemos al navegador. Luego definimos la función getServerSideProps, que obtiene los datos de la API. Devolvemos el conjunto de datos creado desde la función async y lo pasamos al NextPage dentro de las propiedades de la página.

Navega a la nueva página en http://localhost:3000/names-ssr. Deberías ver la lista de nombres de usuario.

Generación de sitios estáticos

SSG crea los archivos HTML solo una vez y los reutiliza para cada solicitud. Es la opción recomendada, porque las páginas pre-renderizadas son fáciles de almacenar en caché y rápidas de entregar. Por ejemplo, una red de entrega de contenido puede captar fácilmente tus archivos estáticos.

Por lo general, las aplicaciones SSG tienen un menor tiempo para el primer renderizado, o el tiempo que tarda después de que un usuario solicita la página (por ejemplo, haciendo clic en un enlace) hasta que el contenido aparece en el navegador. SSG también reduce el tiempo de bloqueo, o el tiempo que tarda el usuario en poder interactuar con el contenido de la página. Buenos puntajes en estas métricas indican un sitio web receptivo, y son parte del algoritmo de puntuación de Google. Por lo tanto, estas páginas tienen un mayor ranking SEO.

Si tu página depende de datos externos, aún puedes usar SSG exportando una función async adicional, getStaticProps, desde el archivo de la página.

Next.js llama a esta función en el momento de la construcción, pasa los datos obtenidos al argumento props de la página y pre-renderiza la página con SSG. Por supuesto, esto solo funciona si los datos externos no son dinámicos.

Intenta crear la misma página que en el ejemplo de SSR, esta vez con SSG. Agrega un nuevo archivo, names-ssg.tsx, en la carpeta pages y luego pega el siguiente código:

```
import type {
  GetStaticProps,
  GetStaticPropsContext,
  InferGetStaticPropsType,
  NextPage,
  PreviewData,
} from "next";
import {ParsedUrlQuery} from "querystring";
import {fetchNames}from "../utils/fetch-names";

type responseItemType = {
  id: string,
  name: string,
};

const NamesSSG: NextPage = (props: InferGetStaticPropsType<typeof getStaticProps>) => {

  const output = props.names.map((item: responseItemType, idx: number) => {
    return (
<li key={name-${idx}}>
          {item.id} : {item.name}
        </li>
    );
  });

  return (
    <ul>
      {output}
    </ul>
```

```
  );
};

export const getStaticProps: GetStaticProps = async (
  context: GetStaticPropsContext<ParsedUrlQuery, PreviewData>
) => {

  let names: responseItemType[] | [] = [];
  try {
    names = await fetchNames();
  } catch (err) {}

  return {
    props: {
      names
    }
  };
};
export default NamesSSG;
```

Aquí solo necesitamos cambiar el código específico de SSR para usar SSG. Por lo tanto, exportamos getStaticProps en lugar de getServerSideProps y ajustamos los tipos en consecuencia.

Cuando visites la página, debería verse similar a la página de SSR. Pero en lugar de solicitar datos frescos en cada visita a http://localhost:3000/names-ssg, los datos se solicitan solo una vez, en la construcción de la página.

Regeneración estática incremental

ISR es un híbrido de SSG y SSR que se ejecuta puramente en el lado del servidor. Genera el HTML en el servidor durante la construcción inicial y envía este HTML pre-generado la primera vez que se solicita una página. Después de que haya pasado un tiempo especificado, Next.js obtendrá los datos y regenerará la página en el servidor en segundo plano. En el proceso, invalida la caché interna del servidor y la actualiza con la nueva página. Cada solicitud subsiguiente recibirá la página actualizada. Al igual que SSG, ISR es menos costoso que SSR y aumenta el ranking SEO de una página.

Para habilitar ISR en páginas SSG, necesitamos agregar una propiedad para revalidate en el objeto de retorno de getStaticProps. Definimos la validez de los datos en segundos, como se muestra en el siguiente listado.

```
return {
      props: {
      names,
      revalidate: 30
   }
};
```

Agregamos la propiedad revalidate con un valor de 30. Como resultado, el servidor personalizado de Next.js invalidará el HTML actual 30 segundos después de la primera solicitud de página.

Renderizado del lado del cliente

Un enfoque completamente diferente, el renderizado del lado del cliente, implica primero generar el HTML con SSR o SSG y enviarlo al cliente. Luego, el cliente obtiene datos adicionales en tiempo de ejecución y los renderiza en el DOM del navegador. El renderizado del lado del cliente es una buena opción cuando se trabaja con conjuntos de datos altamente flexibles y en constante cambio, como los precios de acciones o divisas en tiempo real.

Otros sitios lo utilizan para enviar una versión esquelética de la página al cliente y luego mejorarla con más contenido. Sin embargo, el renderizado del lado del cliente reduce tu rendimiento SEO, ya que sus datos no pueden ser indexados.

El siguiente listado muestra la página que creamos anteriormente, configurada para renderizado del lado del cliente. Crea un nuevo archivo, names-csr.tsx, en la carpeta pages y luego añade el código.

```
import type {NextPage} from "next";
import {useEffect, useState} from "react";
import {fetchNames} from "../utils/fetch-names";
type responseItemType = {
   id: string;
   name: string;
};

const NamesCSR: NextPage = () => {
   const [data, setData] = useState<responseItemType[] | []>();
   useEffect(() => {
```

```
    const fetchData = async () => {
      let names;
      try {
        names = await fetchNames();
      } catch (err) {
  console.log("ERR", err);
      }
      setData(names);
    };
    fetchData();
  });

  const output = data?.map((item: responseItemType, idx: number) => {
    return (
      <li key={name-${idx}}>
        {item.id} : {item.name}
      </li>
    );
  });

  return (
    <ul>
      {output}
    </ul>
  );

};

export default NamesCSR;
```

Este código difiere significativamente de los ejemplos anteriores. Aquí importamos los hooks useState y useEffect. Este último obtendrá los datos después de que la página ya esté disponible. Tan pronto como la función fetchNames devuelva los datos, usamos el hook useState y la variable de estado reactiva data para actualizar el DOM del navegador.

No podemos declarar el hook useEffect como una función async, porque devuelve un valor indefinido o una función, mientras que una función async devuelve una promesa, y por lo tanto TSC lanzaría un error. Para evitar esto, necesitamos envolver la llamada await en una función async, fetchData, y luego llamar a esa función.

La página configurada para renderizado del lado del cliente debería verse similar a las otras versiones. Pero cuando visites http://localhost:3000/names-csr, podrías ver un destello blanco. Esta es la página esperando la solicitud de API asincrónica.

Para obtener una mejor comprensión de los diferentes tipos de renderizado, modifica el código para cada ejemplo en esta sección para usar la API https://www.usemodernfullstack.dev/api/v1/now, que devuelve un objeto con la marca de tiempo de la solicitud.

Exportación HTML estática

El comando next export genera una versión HTML estática de tu aplicación web. Esta versión es independiente del servidor web basado en Node.js incorporado de Next.js y puede ejecutarse en cualquier infraestructura, como un servidor Apache, NGINX o IIS.

Para usar este comando, tu página debe implementar getStaticProps, como en SSG. Este comando no admitirá la función getServerSideProps, ISR o rutas de API.

Ejercicios

Ejercicio 1: Uso del componente next/head

Teoría: Explica cómo el componente next/head permite personalizar la etiqueta <head> de una página en Next.js. Investiga cómo se pueden agregar títulos, meta etiquetas y scripts externos de manera dinámica.

Práctico:

1. Crea una página que use el componente next/head para añadir un título dinámico basado en el nombre de la página actual.

2. Añade una meta etiqueta de descripción personalizada y un enlace a una hoja de estilos externa dentro de la sección <head> de la página.

Ejercicio 2: Navegación con next/link

Teoría: Describe cómo funciona el componente next/link en Next.js para la navegación entre páginas. Explica las ventajas de usar este componente en comparación con enlaces HTML tradicionales.

Práctico:

1. Crea una barra de navegación que permita moverse entre tres páginas diferentes usando next/link.

2. Implementa enlaces que pasen parámetros a través de la URL y muéstralos en las páginas de destino.

Ejercicio 3: Optimización de imágenes con next/image

Teoría: Explica cómo el componente next/image optimiza automáticamente las imágenes en Next.js. Detalla las características como la carga diferida (lazy loading), tamaños adaptativos y formatos optimizados.

Práctico:

1. Implementa una galería de imágenes que use next/image para optimizar las imágenes.

2. Ajusta las propiedades como width, height, layout, y priority para observar cómo afectan la carga de imágenes en dispositivos móviles y de escritorio.

Ejercicio 4: Pre-renderizado y publicación

Teoría: Explica las diferencias entre el renderizado estático y el renderizado del lado del servidor en Next.js. Describe cómo cada opción afecta la velocidad de carga y el SEO de una página web.

Práctico:

1. Crea una página que se pre-renderice usando la función getStaticProps y otra usando getServerSideProps.

2. Publica el sitio y verifica las diferencias en el tiempo de carga y en cómo se indexan las páginas en los motores de búsqueda.

Ejercicio 5: Generación de sitios estáticos y regeneración estática incremental

Teoría: Describe qué es la generación de sitios estáticos (SSG) y la regeneración estática incremental (ISR). Explica cuándo es conveniente utilizar cada técnica y cómo optimizan el despliegue de aplicaciones.

Práctico:

1. Implementa un blog que se genera estáticamente usando getStaticProps con soporte de ISR para actualizar las páginas de artículos automáticamente cada cierto tiempo.

2. Configura la regeneración de los datos del blog cada 10 segundos.

Ejercicio 6: Renderizado del lado del cliente

Teoría: Explica el concepto de renderizado del lado del cliente (CSR) en Next.js y sus diferencias respecto al renderizado del lado del servidor (SSR) y la generación estática (SSG).

Práctico:

1. Crea una página que utilice CSR para cargar datos de una API utilizando useEffect y fetch dentro de un componente React.

2. Muestra cómo la página primero renderiza sin datos y luego actualiza el contenido dinámicamente cuando la llamada a la API se completa.

Ejercicio 7: Exportación HTML estática

Teoría: Describe cómo Next.js permite exportar sitios estáticos completamente en HTML. ¿Qué limitaciones tiene esta técnica en comparación con otros métodos de despliegue de Next.js?

Práctico:

1. Configura un proyecto de Next.js para que exporte su contenido a archivos HTML estáticos usando next export.

2. Verifica que el sitio generado se pueda servir en un servidor estático sin la necesidad de un backend o servidor Node.js.