Actividad 11: Refactorización de Aplicaciones Express.js y React a Next.js

Objetivo: Refactorizar aplicaciones anteriores de Express.js y React en una aplicación Next.js. Esto incluye la migración de rutas API, el manejo de rutas dinámicas, la creación de interfaces personalizadas en TypeScript, y la creación de páginas con componentes personalizados de Next.js.

Vamos a refactorizar las aplicaciones React y Express.js de los aplicaciones en una aplicación Next.js. Como primer paso, resumiremos la funcionalidad que necesitamos construir. La aplicación tiene una ruta de API, api/names, que devuelve nombres de usuario, y otra ruta de API, api/weather/:zipcode, que devuelve un objeto JSON estático y el parámetro de URL. Lo usamos para entender las URLs dinámicas. Además, creamos páginas en /hello y component/weather.

A lo largo de esta clase, ya hemos refactorizado estos varios elementos para que funcionen con el estilo de enrutamiento de Next.js. En este ejercicio, lo reuniremos todo. Sigue los pasos para inicializar la aplicación Next.js. Dentro de la carpeta sample-next, nombra tu aplicación refactored-app.

Almacenando interfaces y tipos personalizados

Creamos un nuevo archivo, custom.d.ts, en la raíz del proyecto para almacenar las definiciones de interfaz y tipos personalizados. Es similar al que usamos a las actividades de typeScript y React. La principal diferencia es que agregamos tipos personalizados para la aplicación Next.js.

```
interface WeatherProps {
   weather: string;
}
type WeatherDetailType = {
    zipcode: string;
   weather: string;
   temp?: number;
};
type responseItemType = {
 id: string;
name: string;
};
```

Usaremos la interfaz personalizada WeatherProps para el argumento props de la página que muestra los componentes meteorológicos, components/weather. El tipo WeatherDetailType es para la ruta de API api/weather/:zipcode, que utiliza un código postal obtenido dinámicamente. Finalmente, usamos responseItemType en la ruta de API api/names para tipar la respuesta de la obtención.

Creando las rutas de API

A continuación, volvemos a crear las dos rutas de API del servidor Express.js. Las secciones anteriores de las notas de clase mostraron este código refactorizado. Para la ruta api/names, crea un nuevo archivo, names.ts, en la carpeta api, luego agrega el código:

```
import type { NextApiRequest, NextApiResponse } from "next";

type responseItemType = {

    id: string;

    name: string;

};

export default async function handler(

  req: NextApiRequest,

   res: NextApiResponse

): Promise<NextApiResponse<responseItemType[]> | void> {

          const url = "https:// www . usemodernfullstack . dev / api / v1 / users";

          let data;

          try {

           const response = await fetch(url);

           data = (await response.json()) as responseItemType[];

           } catch (err) {

                return res.status(500);

           }

           const names = data.map((item) => {

                     return { id: item.id, name: item.name };

           });

           return res.status(200).json(names);

}
```

Lee las notas de clase para una explicación detallada del código.

Migra la ruta dinámica api/weather/:zipcode del servidor Express.js a la aplicación Next.js creando un archivo [zipcode].js en la carpeta api y añadiendo el código:

```
import type { NextApiRequest, NextApiResponse } from "next";

type WeatherDetailType = {

    zipcode: string;

     weather: string;

      temp?: number;

}

export default async function handler(

      req: NextApiRequest,

      res: NextApiResponse

): Promise<NextApiResponse<WeatherDetailType> | void> {

       return res.status(200).json({

           zipcode: req.query.zipcode,

           weather: "sunny",

           temp: 35

           });

}
```

Lee las notas de clase para una explicación detallada del código.

Creando las rutas de páginas

Ahora trabajamos en las páginas. Primero, para la simple página hello.tsx, creamos un nuevo archivo en la carpeta pages y agregamos el código siguiente.

```
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;
```

Este código renderiza el ejemplo Hello World! y utiliza los componentes personalizados Head, Link e Image de Next.js.

La segunda página es la ruta anidada pages/components/weather.tsx. Como antes, creamos un nuevo archivo, weather.tsx, en una carpeta llamada components, dentro de la carpeta pages. Agrega el código siguiente:

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

import React, { useState, useEffect} from "react";

const PageComponentWeather: NextPage = () => {

    interface WeatherProps {

        weather: string;

              }

            const WeatherComponent = (props: WeatherProps) => {

              const [count, setCount] = useState(0);

            useEffect(() => {

                    setCount(1);

           }, []);

           return (

               <h1 onClick={() => setCount(count + 1)}>
```


```
                El clima es {props.weather},

                Y el contador muestra  {count}

            </h1>

        );

    };

return (<WeatherComponent weather="sunny" />);

};

export default PageComponentWeather;
```

Este listado utiliza los hooks useState y useEffect para crear una interfaz de usuario reactiva. Podemos eliminar la definición de interfaz personalizada para los WeatherProps de este archivo. El archivo custom.d.ts ya los añade a TSC.

Ejecutando la aplicación

Inicia la aplicación con el comando npm run dev. Ahora puedes visitar las mismas rutas que creamos para el servidor Express.js y ver que son funcionalmente las mismas.

Deberias haber creado tu primera aplicación full-stack basada en Next.js. Juega con el código y prueba usar CSS global y de componentes para estilizar tus páginas.

Ejercicios

Ejercicio 1: Agregar manejo de errores en las rutas de API

Objetivo: Mejorar la robustez de la aplicación manejando errores de manera apropiada en las rutas de API.

1. En la ruta `api/names.ts`, modifica el código para manejar errores más detalladamente. En lugar de simplemente devolver un error 500, devuelve un mensaje de error específico con el código de estado adecuado (400, 404, etc.).
2. Haz lo mismo en la ruta `api/weather/:zipcode.ts`, incluyendo un caso en el que el código postal no es válido (por ejemplo, si es un código que no cumple con los requisitos de longitud).

Pregunta teórica: ¿Por qué es importante devolver códigos de estado HTTP adecuados en las API y cómo pueden afectar la experiencia del usuario o el comportamiento de los clientes que consumen la API

Ejercicio 2: Integración de CSS módular

Objetivo: Refactorizar la aplicación para utilizar estilos CSS modulares.

1. Refactoriza las páginas `hello.tsx` y `components/weather.tsx` para que utilicen archivos de estilos CSS modulares. Crea archivos como `Hello.module.css` y `Weather.module.css`.
2. Asegúrate de que cada componente tenga su propio archivo de estilos para mantener los estilos aislados.

Pregunta teórica: Explica las ventajas de usar CSS modular en comparación con el uso de un archivo CSS global en una aplicación grande.

Ejercicio 3: Extender la funcionalidad del contador en weather.tsx

Objetivo: Mejorar la funcionalidad del componente `WeatherComponent`.

1. Modifica el contador del componente `WeatherComponent` para que se guarde en `localStorage`. De esta manera, cuando el usuario recargue la página, el contador mantendrá su valor.
2. Utiliza `useEffect` para verificar si el valor del contador ya está almacenado en `localStorage` al cargar la página.

Pregunta teórica: ¿Cómo puedes usar `localStorage` en una aplicación Next.js para guardar datos de sesión, y cuáles son las limitaciones de esta aproximación?.

Ejercicio 4: Implementar parámetros opcionales en la API

Objetivo: Extender la funcionalidad de la API para soportar parámetros opcionales.

1. Modifica la ruta `api/weather/:zipcode` para que también acepte un parámetro opcional `?tempUnit=metric` o `imperial` y ajuste la respuesta del objeto JSON en consecuencia (por ejemplo, convertir la temperatura de Celsius a Fahrenheit).
2. Actualiza la lógica en el front-end para enviar este parámetro opcional en la petición API desde `components/weather.tsx`.

Pregunta teórica: ¿Cómo gestionaría Next.js rutas con parámetros opcionales y cómo esto podría beneficiar la flexibilidad de la API?.

Ejercicio 5: Agregar SEO con el componente `next/head`

Objetivo: Mejorar el SEO de la aplicación con meta tags dinámicos.

1. Modifica el componente `Hello` para que los meta tags dentro del componente `Head` se configuren dinámicamente dependiendo del contenido de la página.

2. Agrega meta tags que incluyan descripciones, palabras clave, y otras propiedades que mejoren el SEO de la aplicación.

Pregunta teórica: ¿Por qué es importante el SEO en una aplicación web y cómo afecta la indexación de los motores de búsqueda al rendimiento de la misma?.

Ejercicio 6: Implementar pre-renderización incremental en la página weather

Objetivo: Utilizar las capacidades de pre-renderizado incremental de Next.js.

1. Refactoriza la página `components/weather.tsx` para que utilice `getStaticProps` junto con regeneración estática incremental (ISR). Configura que los datos del clima se actualicen automáticamente cada 30 segundos.
2. Comprueba que la regeneración estática funcione correctamente haciendo cambios en los datos de la API.

Pregunta teórica: ¿Cuáles son las principales diferencias entre SSR (Server-Side Rendering) y ISR (Incremental Static Regeneration) en términos de rendimiento y casos de uso?