PROYECTO BLOG CRUD

Adrián Page Fernández

Uso (adjunto al proyecto en el readme)

Prueba Blog CRUD

Pasos para arrancar el proyecto

Base de datos

En phpMyAdmin, crear una base de datos llamada "page-blog" e importar el archivo page-blog.sql de este repositorio.

Dicho archivo se encuentra en el directorio original, junto a este archivo README.md

Terminales

Una vez instalada la bbdd y clonado el repo recomiendo abrir la terminal y dividirla en dos una para Backend Y otra para Frontend

Backend

Accedemos al directorio Backend

```
cd page-blog/Backend
```

Instalamos las dependencias:

```
npm i
```

Arrancamos el servidor con uno de estos dos comandos:

```
node index.js

nodemon
```

(recomiendo el segundo por si se efectúa algún cambio o prueba)

Frontend

Accedemos al directorio Frontend

```
cd page-blog/front
```

Instalamos las dependencias:

```
npm i
```

Arrancamos la aplicación de React:

```
npm start
```

Presentación del proyecto

-Para este blog sencillo con sistema CRUD creado con react he creado una vista principal o Homeview que funciona como feed de los post. Se lanza automáticamente y se puede acceder a ella por la navbar con el botón "inicio"

```
```

-Para añadir una publicación o post en el blog accedemos desde el botón del navbar "añadir publicación". Esto nos hará acceder a la vista addpostview, la cual tiene un formulario de creación de post con título, contenido e imágenes. La imagen adjuntada será copiada a backend dentro de public/images con un id correspondiente al post, luego veremos qué pasa al borrarla. Al crearse se le asigna una fecha de creación automáticamente, esta puede cambiar al ser editado el post, cosa que también veremos a continuación.

```
```

-Volviendo al inicio o home veremos una cierta cantidad de post, incluido el ultimo que hayamos subido, el cual se pondrá el primero en la lista. Si bajamos veremos que la página tiene paginación, de manera que podremos acceder a post antiguos en las demás páginas.

```
```

-Para editar uno de estos post simplemente haremos click sobre su imagen o sobre el texto del título. Estos reaccionan a nuestro ratón agrandándose para mostrar su reactividad. Al hacer click nos llevará a otra vista, la singlepostview. Ahí veremos el post únicamente, sin las demás

publicaciones, más grande y con sus distintos elementos dentro de un contenedor con borde para que se distinga. Si damos al botón de borrar se borrará el post. Si le damos a editar se desplegarán unas áreas de texto en donde escribir los cambios al título o al contenido y un botón para guardar los cambios y otro para cancelarlos. La imagen puede dejarse como está o cambiarse. Si se cambia, la nueva imagen, se guarda automáticamente en el backend y se borra la antigua. Al guardar los cambios también se actualiza la fecha de última actualización del post.

```

-Borrar el post. Se puede desde la vista del post único o desde el home, donde podemos darle a borrar a cada post pues cada uno trae consigo un botón rojo que así lo indica. Al hacerlo no solo se borra el texto de la base de datos sino también la imagen asociada, para evitar que quede huérfana.

```

-El diseño lleva bootstrap para hacerlo responsivo y alguna mediaquery donde me pareció más conveniente. La barra de navegación se contrae en un pequeño menú hamburguesa para pantallas pequeñas.

Planteamiento

Mi idea para implementar el proyecto fue la siguiente:

Empezar por el Backend y la base de datos y después hacer el Front y el maquetado. Así me ahorraba poner demasiadas cosas que luego quizás desechase y algunos problemas de elementos que se chocan entre sí y rompen funciones al cambiarlos.

Utilicé esta página para hacerme una idea del aspecto que tendría el blog *app.moqups.com.*

Recursos

- Visual Code Studio: VS Code es un editor de código fuente ligero pero muy potente.
- Bootstrap: es el framework más popular del mundo a la hora de crear páginas web que sean responsive. Como además lo había utilizado anteriormente y Tailwind no, pues me decanté por este. En el proyecto se usó React-Bootstrap, una biblioteca que ha reconstruido Bootstrap para ser utilizado en React, con sus propias plantillas y usos.
- Postman: una herramienta de Chrome que permite crear, probar y usar APIs de forma sencilla y rápida. Muy útil para comprobar el funcionamiento de mis endpoints.
- Consola de Microsoft edge.
- Xampp: Una distribución de Apache para gestión de MySQL. También incluye una herramienta llamada phpMyAdmin, que sirve para administrar bases de datos MySQL a través de una página web. Muy útil por su interface para visualizar tablas y columnas.
- Git y Github Desktop: Para conectar con el repositorio online del trabajo crear ramificaciones del proyecto en que probar cosas y luego unirlas a la rama principal.

Realización

Aquí comentaré el desarrollo del proyecto de manera general, comentando algunos problemas y como los resolví. Puede que se me pase algún detalle porque la cantidad de pruebas que hice fue ingente, pero debería bastar para hacerse una idea del proceso que tuvo lugar. También omitiré detalles técnicos del tipo "instalé *body-parser* y *cors* en el back porque si no me da error el terminal" y similares.

Primero hice la bbdd desde PHPMyAdmin. La llamé page-blog (no me compliqué mucho con el nombre la verdad) con una única tabla posts y cinco columnas (id, contenido, titulo, el url de la imagen y la fecha)

A continuación en Visual Studio, tras entrar en una carpeta para proyectos que tengo vacía, instalé el proyecto react desde la terminal con: npx create-react-app page-blog

Una vez instalado lo dividí en dos. Una parte en la carpeta Backend y otra Frontend, dejando fuera sitio para cuando exportase la base de datos y el readme con las notas de uso. Tambien puse gitignore para evitar que se guardasen en git la cantidad abrumadora de archivos del node_modules.

Backend

Empecé en el Backend, instalando node, express y sequelize y creando una conexión a la base de datos. Para ello hice un archivo connection.js en la carpeta db (database). Importé sequelize, definí la ruta de conexión localhost 3306 y con un sistema de promesas autentiqué la conexión.

Luego en app.js, que sería el archivo principal del Backend, cree la instancia de express, el enrutador de publicaciones y que el servidor se lanzase en el puerto 3000. También puede observarse un enrutamiento para el archivo de imágenes public/images. Más tarde lo explicaré.

Con el puerto listo cree una carpeta de rutas o routes y dentro metí los endpoints post.js. En principio fueron dos métodos get, uno para traer todos los post y al home, otro definido para traer los post solo por id. A continuación, un método post para subir propiamente un post, un put para actualizarlo por la id y un delete para borrarlo, también según su id. Todos fueron comprobados mediante Postman.

Primer obstáculo:

Llegados aquí toca hablar de Multer, una biblioteca de midleware que apliqué tiempo después cuando vi que había que guardar las imágenes de los post en el proyecto. Hasta ahora nunca había hecho nada así, y cuando ponía imágenes para base de datos solía sacar su url directamente de internet. Como la consigna pedía guardarlos pregunté a compañeros, en general con el mismo problema y uno propuso usar este sistema, básicamente para el almacenaje.

A Multer le indiqué un destino y un sistema de nombrado de imágenes con la fecha, por si duplicaba imágenes que me las cogiera igual. Luego con upload.single('image') aplicado en los métodos post y put conseguí que efectivamente subiera las imágenes a donde quería.

Posiblemente todo este proceso fue el más complejo del proyecto por mi desconocimiento, por suerte encontré documentación.

Obviamente había que poder borrar las imágenes una vez subidas, ahí encontré Fs, en la propia documentación de Multer recomendada para la tarea. Básicamente lo que hace es construir una ruta de la imagen sacándola por el nombre del archivo en la bbdd y mediante fs.unlinkSync() se puede eliminar. Esto se aplicó al delete y también al put (para borrar una imagen cuando se adjunta una nueva al editar un post).

Hecho esto terminamos nuestra visita al Backend.

Frontend

En el Frontend empecé creando un sistema de archivos, como Styles para los css, Components para los componentes y Views para las vistas. Los rellené con archivos vacios de lo que iba a ser la página, como una vista home, una de edición de post, un elemento header, un footer, etc.

Después alteré el punto de entrada que crea react de base, el index.js metiéndole un react-dom e importé bootstrap a sabiendas que luego lo utilizaría mucho.

Pasé a app.js, donde definiría todas las rutas del proyecto, renderizar las vistas según esas rutas y una serie de componentes comunes a todo el blog como el header, el navbar y el footer.

Luego hice los componentes, muy sencillos para usarlos como guias, más tarde añadí una estructura de boostrap para hacerlos responsivos y darles estilos css. No los explicaré al pormenorizado solo los dos que considero más complejos, el editpost.js y el addpostform.js, pero más adelante.

Creé unas vistas sobre las que colocar los componentes y que serían el equivalente a la distintas páginas de una web. En este caso solo necesité 3. La del home o homeview, una para crear post o addpostview y una para ver un único post y editarlo, la singlepostview.

Homeview renderiza la lista de publicaciones o post con la paginación necesaria para navegar entre ellas. Primero se importan una serie de dependencias como useEffect y useCallback así como el componente post. Dentro de la función, se definen varios estados con la ayuda del hook useState. Posts es un estado que almacena la lista de publicaciones. CurrentPage es un estado que almacena el número de la página actual. TotalPages es un estado que almacena el número total de páginas.

El fetchpost se define utilizando el hook useCallback para garantizar que su referencia no cambie en cada renderizado. La función realiza una solicitud a una URL utilizando fetch y obtiene los datos en formato .JSON. Luego, se desestructura el objeto “data” para extraer las propiedades currentPage, totalPages y posts. Los valores obtenidos se utilizan para actualizar los estados utilizando las funciones setPosts, setCurrentPage y setTotalPages. Si ocurre algún error, se muestra en la consola.

Debajo hay una función para borrado d epost handleDelete. Hace una solicitud de eliminación a una URL específica para el ID de la publicación. Después de la eliminación exitosa, se llama a fetchPosts para actualizar la lista de publicaciones.

La función handlePageChange se utiliza para cambiar la página actual. Actualiza el estado currentPage con el número de página proporcionado.

Con el return renderizamos el componente post. El contenido principal se encuentra dentro de un contenedor div con la clase "homeview-container".

A continuación, se itera sobre la lista de publicaciones utilizando el método map. Para cada publicación, se crea un componente Post pasando la publicación como una propiedad (post) y la función handleDelete como onDelete.

Después de las publicaciones, se muestra la sección de paginación.

Por último, se exporta el componente Homeview como el valor por defecto del módulo.

La siguiente vista es **AddPostView** a la que se accede por el link del navbar. Esta vista renderiza el formulario de creación de post pasando la función handleSubmit como un prop llamado onSubmit. Esto permite que el formulario comunique los datos de la nueva publicación al componente AddPostView cuando se envía el formulario.

Hablaré aquí un poco de ese componente que ya había nombrado previamente, el AddPostForm. Crea una función donde se utilizan hooks de estado useState para manejar los valores de diferentes campos del formulario, así como para mostrar un mensaje de error.

El componente define tres funciones de cambio para manejar la actualización de los campos del formulario:

- handlePostTitleChange se ejecuta cuando el valor del campo de título del post cambia. Actualiza el estado postTitle con el nuevo valor.
- handlePostContentChange se ejecuta cuando el valor del campo de contenido del post cambia. Actualiza el estado postContent con el nuevo valor.
- handleImageFileChange se ejecuta cuando se selecciona un archivo en el campo de imagen. Actualiza el estado imageFile con el archivo seleccionado.

La función handleSubmit se ejecuta cuando se envía el formulario. Se utiliza el método preventDefault para evitar que el formulario se envíe automáticamente.

Se valida si los campos postTitle, postContent e imageFile tienen valores. Si alguno de ellos falta, se establece un mensaje de error en el estado errorMessage y se retorna para evitar enviar el formulario.

Si todos los campos tienen valores, se limpia el mensaje de error setErrorMesage. Se crea un objeto FormData para enviar los datos del formulario al servidor. Los valores de postTitle, postContent e imageFile se agregan al objeto FormData.

Se realiza una solicitud post al servidor utilizando la función fetch. Se envía el objeto FormData como el cuerpo de la solicitud. Se espera la respuesta del servidor y se obtiene los datos de la respuesta utilizando response.json. Si la respuesta es exitosa se muestra en la consola el ID de la nueva publicación creada. Si la respuesta no es exitosa, se muestra en la consola el error de creación de la publicación data.error.

En la función return, se renderiza el formulario de añadir una publicación. El formulario se envía mediante el evento onSubmit, que llama a la función handleSubmit. Luego se renderizan los campos del formulario: el campo de título del post (<input>), el campo de contenido del post (<textarea>) y el campo de imagen (<input type="file">). Finalmente, se muestra un botón para enviar el formulario (<button type="submit">).

Y por último se exporta el componente.

Por ultimo tenemos la vista **SinglePostView** para ver un único post y editarlo. Esta parte tambien fue problemática por todo el juego de estados para la edición, como veremos a continuación.

A esta vista se accede también por link, con el router react dom, al hacer click en la imagen del post o su título.

Primero se importan una serie de hooks como useParams o useState.

Se define la funcion SinglePostView que utiliza hooks de estado useState para manejar el estado de la publicación y el modo edición.

Se utiliza el hook useParams para obtener el ID de la publicación de los parámetros de la URL. Luego el componente define una función fetchPost utilizando el hook useCallback que realiza una solicitud GET al servidor para obtener la publicación correspondiente al ID. Cuando se obtiene la respuesta, se actualiza el estado post con los datos recibidos.

El componente utiliza dos efectos secundarios (useEffect) para realizar acciones en determinados momentos: el primer efecto se ejecuta una vez al cargar el componente y llama a la función fetchPost para obtener la publicación inicial. El segundo efecto se ejecuta cuando el estado shouldFetchPost cambia. Si se debe buscar una nueva publicación (shouldFetchPost es true), se llama a la función fetchPost nuevamente para obtener la publicación actualizada. Luego, se establece shouldFetchPost en false.

El componente tiene una función toggleEditMode que se ejecuta cuando se hace clic en el botón "editar". Activa el modo edición.

El componente tiene una función handleDelete que se ejecuta cuando se hace clic en el botón "borrar". Esta función realiza una solicitud DELETE al servidor para eliminar la publicación correspondiente al ID.

El componente tiene también una función handleSave que se pasa al componente EditPost y se ejecuta cuando se guarda la publicación editada. Esta función recibe la publicación editada como argumento, muestra los datos en la consola, cambia el modo de edición (toggleEditMode) y establece shouldFetchPost en true para buscar la versión actualizada de la publicación. Si la publicación no se ha cargado todavía (post es null), se muestra un mensaje de "Cargando...".

Si la publicación está disponible, se muestra la vista detallada de la publicación con los siguientes elementos: si isEditMode es true, se muestra el componente EditPost en lugar de la vista detallada. Si isEditMode es false, se muestran el título, la imagen, el contenido y la fecha de actualización de la publicación. Además, se muestran botones para editar y borrar la publicación.

Respecto al componente que renderizamos EditPsot y que ya nombramos antes vamos a hacer un pequeño recorrido de cómo funciona.

Hacemos las importaciones de rigor. Luego EditPost, que es una función que toma dos propiedades como argumentos: post, que contiene los datos de la publicación a editar, y dos funciones de devolución de llamada, onSave y onCancel, utiliza el hook useState para manejar el estado de la publicación editada.

El componente tiene dos funciones, handleEditChange y handleImageFileChange, que se utilizan para manejar los cambios en los campos de edición. Estas funciones actualizan el estado editedPost con los valores ingresados por el usuario.

El componente tiene una función handleSave que se ejecuta cuando se hace clic en el botón "Guardar cambios". Esta función crea un objeto FormData con los datos de la publicación editada y realiza una solicitud PUT al servidor para guardar los cambios. Si la solicitud es exitosa, se llama a la función onSave proporcionada como prop para indicar que se han guardado los cambios.

El componente renderiza un formulario de edición con los campos post_title, image y post_content. Los valores de los campos se obtienen del estado editedPost y se actualizan mediante las funciones de cambio correspondientes.

El componente permite al usuario seleccionar una imagen para adjuntar a la publicación. Cuando se selecciona un archivo, se llama a la función handleImageFileChange para actualizar el estado editedPost con el archivo seleccionado. Si no se introduce imagen deja la que teníamos previamente, de lo contrario indicará al Backend que borre la imagen previa e instale la nueva en public/images.

El componente también tiene un botón que ejecuta onCancel para cancelar la edición.

Visto esto ya tenemos una idea de cómo funciona la página. Respecto a la maquetación estilos y demás, como ya adelantaba utilicé Boostrap. La importé directamente en index, mi punto de entrada del proyecto para no tener que hacerlo en cada elemento. Luego apliqué bootstrapa todos los renderizados en componentes y vistas allá donde fuese necesari,o, dándole clases y estableciendo rutas a los archivos css. Hay uno por cada componente y vista salvo footer que era tan poco código que lo encontré innecesario. Al hacerlo así en lugar de un único archivo de estilos, el típico app.css, me resultó más sencillo aplicar pequeños cambios a las partes y leer el código.

El estilo de la página, tanto en color como imágenes es de temática veraniega y comfy, por así decir. Utilizando varias imágenes de playas en mitad de la noche o el ocaso, generadas por ia (como un guiño a Guillermo que sé le encanta ese tema), el footer y el navbar tienen un negro desaturado para diferenciarse del fondo como elementos ajenos mientras que el header trata de continuar con el tono azulado oscuro del fondo como si fuera el cielo extendiéndose más allá.

El propio Boostrap lleva un sistema muy responsivo de base, aunque por si acaso metí algo de mediaquerys en alguna vista para que los elementos saliesen mejor. No explicaré el pormenorizado de cada estilo aplicado, pero por ejemplo y por acabar la parte de diseño responsivo comentaré algo del navbar y su menú hamburguesa.

Se utiliza el hook useState para definir el estado del menú hamburguesa. expanded es una variable de estado que indica si el menú hamburguesa está expandido o no. Al inicio, se establece en false, lo que significa que el menú hamburguesa estará colapsado inicialmente.

El toggleNavbar es una función que se ejecuta cuando se hace clic en el menú hamburguesa. Cambia el estado expanded al valor opuesto utilizando setExpanded(!expanded). Esto permite alternar entre la expansión y el colapso del menú hamburguesa.

utilizando <Navbar> de react-bootstrap se establecen las propiedades bg="dark" y expand="lg" para especificar el fondo oscuro y la expansión para pantallas grandes.

El componente Navbar.Toggle crea el menú hamburgués propiamente. Se establece la propiedad onClick para que llame a la función toggleNavbar cuando se haga clic en el menú hamburguesa. Además, se establece la propiedad bg="light" para cambiar el color de fondo del menú hamburguesa a claro. Dentro de <Navbar.Toggle>, se coloca el ícono de hamburguesa personalizado <FaBars>. Al principio no usé iconocs pero por algún motivo el que venía de base de boostrap me estaba dando problemas con el fondo así que añadí estos.

En Navbar.Collapse envolvemos el contenido del menú de navegación que se mostrará cuando el menú hamburguesa esté expandido. Aquí se colocan los enlaces de navegación. Estos usan Nav.Link de react-router-dom. Al hacer clic en estos enlaces, se ejecuta la función toggleNavbar, lo que hará que el menú hamburguesa colapse.

El resto de colores fuentes y demás están linkeados al css del mismo nombre.

Nota Final

Espero estén conformes con el blog y la memoria. Nunca antes había hecho una así que no sé si me habré excedido o quedado corto con la explicación, aunque entiendo que lo que buscan es una explicación a grandes rasgos de cómo se planteó y realizó el proyecto y con esto se pueden hacer una idea. Espero que este pequeño blog les sea de su agrado y si tienen alguna pregunta no duden en contactarme.

Muchas gracias por leerme 😉.