**Manual Técnico**

**Sign up**

Se usaron los siguientes archivo

* Signup.js
* Signup.php

Paralo que es **Signup.js**, se empezó de la siguiente manera:  
Texto

Descripción generada automáticamente

Empezado a importar las librerías necesarias en la cabecera para que el proyecto funcionara

* **React**: Se importa la librería base de React, que es necesaria para crear componentes.
* **useState**: Es un hook de React que permite manejar el estado local dentro de un componente funcional.
* **Link**: Se usa para crear enlaces de navegación. Hook de react-router-dom
* **axios**: Es una librería que facilita la realización de solicitudes HTTP desde el frontend a un backend
* **useNavigate**: Hook de react-router-dom que permite la navegación programática entre rutas.

Texto

Descripción generada automáticamente

Posteriormente tenemos el estado inicial del formulario, por lo que hace uso del hook useState, en donde inicializamos el estado **values**, el cual contiene los valores de los campos del formulario. **setValue** que se usará para actualizar ese estado cuando el usuario escriba los campos pertinentes.

**Valores predeterminados:**

* **name**: Cadena vacía para el campo de nombre.
* **lastname**: Cadena vacía para el apellido.
* **email**: Cadena vacía para el correo electrónico.
* **password**: Cadena vacía para la contraseña.
* **id\_rol**: Por defecto, se establece como 1, que representa un "User" (usuario).



Nos va ayudar a poder redirigir a otra pagina

Texto

Descripción generada automáticamente

Esta función se activa cada vez que el usuario escribe en un campo, por lo que Actualiza el estado **values** con el nuevo valor ingresado.

* Evento.target.name: Hará referencia al nombre del campo del formulario que está siendo actualizado, como lo que es name, lastname, email, etc.
* Evento.target.value: Es el valor ingresado por el usuario

Por ejemplo, si el usuario escribe en el campo de email, evento.target.name será email, y el estado se actualizará con el valor ingresado.

Texto

Descripción generada automáticamente

Evento.preventDefault(): Maneja el envío del formulario, evitando el comportamiento predeterminado del formulario, ósea que recarga la página, oseá que el formulario no se envía de manera tradicional, sino que podemos hacer control sobre su envió.

Luego en Axios.post realiza la solicitud HTTP de tipo POST al archivo signup.php, enviando los datos **values** al servidor. Si es exitosa, pasa lo siguiente:

* **.then()**: Recibe la repuesta del servidor y si el usuario fue creado con éxito, muestra una alerta y redirige al usuario a la pagina principal por medio de **navigate**

Si ocurre un error

* **.catch()**: Captura el error y muestra un mensaje de alerta

**Renderización del formulario**

Texto

Descripción generada automáticamente

Inicia el formulario HTML, el atributo **onSubmit={handleSubmit}** especifica que cuando el usuario envíe el formulario, se ejecutará la función **handleSubmit**.

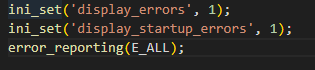
Texto

Descripción generada automáticamente

* Label etiqueta del campo de entrada
* Input es el campo de entrada de texto
* ‘name’, especifica el nombre del campo
* onChange={handleInput}: Se ejecuta cuando el usuario escribe en el campo, lo que actualiza el estado, onChange es un manejador de eventos que ayuda para gestionar los cambos en los campos de formulario.
* value={values.name}: Vincula el valor del campo con el estado **values.name**



El botón tiene como atributo type='submit’, lo que significa es que al hacer click, intentará enviar el formulario al que está asociado, ósea se dispara al evento onSubmit, que está enlazado con la función, handleSubmit.

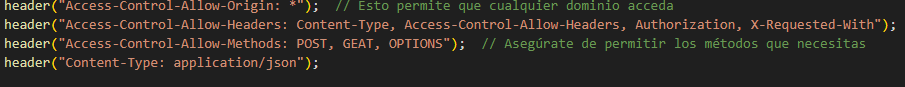
Paralo que es **Signup.php**, se empezó de la siguiente manera:  


Habilito poder tener la visualización de errores para la depuraci´n

**ini\_set('display\_errors', 1):** Activa la visualización de errores en PHP.

**ini\_set('display\_startup\_errors', 1**): Muestra los errores que puedan ocurrir durante el arranque de PHP.

**error\_reporting(E\_ALL)**: Establece que todos los errores, advertencias y avisos deben ser reportados.



Habilito los CORS para permitir las solicitudes desde el frontend.

**Access-Control-Allow-Origin**: Permite que cualquier dominio pueda hacer solicitudes al servidor. Permite que el frontend (React, en este caso) pueda comunicarse con el backend (PHP).

**Access-Control-Allow-Headers**: Especifica los encabezados permitidos en las solicitudes.

**Access-Control-Allow-Methods**: Permite solicitudes de tipo POST, GET y OPTIONS al servidor.

**Content-Type: application/json**: Especifica que la respuesta del servidor será en formato JSON.

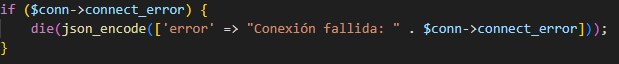
Texto

Descripción generada automáticamente

Configuración de la base de datos.



**$conn = new mysqli(...):** Establece una conexión a la base de datos utilizando **mysqli**, que es una extensión de PHP para interactuar con bases de datos.



Este verifica la conexión de la base de datos

* **$conn->connect\_error:** Comprueba si hay algún error en la conexión.
* **die(json\_encode(...)):** Si hay un error de conexión, termina la ejecución del script y devuelve un mensaje de error en formato JSON.



**file\_get\_contents("php://input"):** Lee los datos que fueron enviados en la solicitud POST, ósea los valores del formulario. Nos referimos a la solicitud HTTP que el front envía al servidor para **enviar los datos.**

**json\_decode(..., true):** Decodifica los datos JSON en un array asociativo de PHP.

Texto

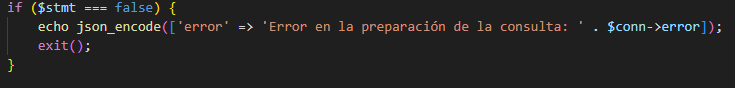
Descripción generada automáticamente

Aquí se asignan los valores recibidos a las variables de PHP para usarlas más adelante. Estos son los datos del usuario que se van a insertar en la base de datos.



**$sql:** Se define una consulta SQL para insertar un nuevo usuario en la tabla **User** con los campos name, lastname, email, password e id\_rol.

**$conn->prepare($sql):** Prepara la consulta SQL para evitar inyecciones SQL y mejorar la seguridad. El uso de **?** permite que los valores se vinculen.



**$stmt === false:** Comprueba si hubo un error al preparar la consulta SQL.

Si hubo un error, devuelve un mensaje de error en formato JSON y detiene la ejecución del script.



**bind\_param("ssssi", ...):** Vincula los valores a los marcadores ? en la consulta SQL.

**"ssssi":** Especifica los tipos de los valores que se están vinculando:

**s**: String (cadena de texto) para name, lastname, email y password.

**i**: Integer (entero) para id\_rol.

Texto

Descripción generada automáticamente

**$stmt->execute():** Ejecuta la consulta preparada.

Si la inserción es exitosa, se devuelve un mensaje de éxito en formato JSON.

Si falla, se devuelve un mensaje de error junto con el error de MySQL



**$stmt->close():** Cierra la consulta preparada para liberar recursos.

**$conn->close():** Cierra la conexión a la base de datos.

¿Cómo funciona entonces el envió y recibo de datos?

En este caso PHP envía los datos al front, y el fron recibe esos datos para hacer algo con eso. Primero se comienza que el usuario ingresa su email, contraseña, nombre, apellido en el formulario, cuando hace click al botón “SUBMIT”, losdatos se envían a backend utilizando una solicitud HTTP POST con **axios**

Captura de pantalla de computadora

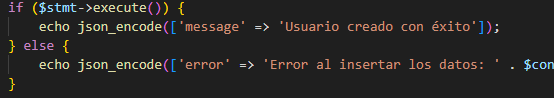
Descripción generada automáticamente

Entonces axios.post envia una solicitud POST al backend con los datos ingresados en **values**. El endpoint al que se envia es el que esta en la imagen, luego si es exitoso o no envia un mensaje.

Ahora bien, PHP recibe la solicitud HTTP POST que contiene los datos del formulario, luego son decodificados de JSON a un array PHP con json\_decode



Luego verifica que los campos requeridos existan, si falta uno manda un error. Posteriormente si esta correcto, el PHP devuelve un mensaje de éxito en formato JSON.



El front recibe la repuesta y la maneja de acuerdo al contenido, si esta bien , muestra el mensaje de éxito, si no, muestra mensaje de error.

Pantalla de computadora con letras

Descripción generada automáticamente con confianza media

**LOGIN**

Para el LOGIN, se usaron los siguientes componentes:

* Login.js
* login.php

Para en la parte del frontend, se uso el Login.js, la cual funciona de esta manera:

Primero hace las importaciones de las librerías necesarias, ya se explicó con anterioridad que hace cada una.

Texto

Descripción generada automáticamente

De igual manera, el **navigate** se usa para redigir al usuario a una ruta diferente, según el rol.

Texto

Descripción generada automáticamente

Ahora bien, se hace uso de **useState**, donde **values** es un objeto que contiene los valores de los campos del formulario que inician como null, luego tenemos **setValues**, en el cual actualiza los valores de values cada vez que el usuario escribe en el campo.

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

Creamos una función que se ejecuta cuando el usuario decide acceder como guest, el rol es guardado en el **localStorage** como guest y redirige al usuario a la página diseñada para guest

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

Ahora bien, comenzamos con el event.preventDefault() el cual nos ayuda a evitar que el formulario se envíe como que recargue la página.



Posteriormente, usamos axios.post() para enviar los valores del formulario al PHP, se envían los valores en formato JSON y se espera de la respuesta del PHP

**data** es el objeto que contiene la información enviada desde tu backend en PHP, cuanod hacemos la solicitud, el PHP procesa la solicitud, realiza la operación (buscar en la base de datos) luego devuelve una respuesta en formato JSON, y se captura en **data**

Si el PHP devuelve un error, se muestra un error con alert(data.error).

Si el PHP devuelve el rol del usuario, se guarda el rol y el nombre, password en el localStorage.

Ahora bien dependiendo, el rol se redirige a la ruta correspondiente. Si ocurre un error durante la solicitud, se captura con **catch** y se muestra un mensaje de alerta.

Texto

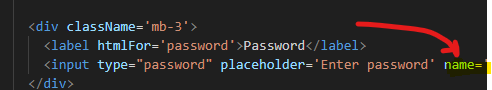
Descripción generada automáticamente

**handleInput**: Esta función se ejecuta cada vez que el usuario escribe en un campo del formulario. Actualiza los valores de **email** y **password** dentro del estado **values**.

**event.target.name** se refiere al atributo **name** del campo que está siendo modificado (email o password), y **event.target.value** es el nuevo valor ingresado.

Texto

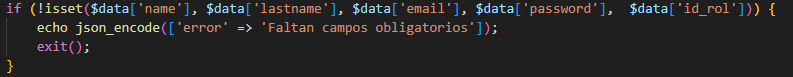
Descripción generada automáticamente



**PHP**



Acá lee el cuerpo de la solicitud POST, ósea los datos enviados desde el frontend, file\_get\_contents("php://input") va a capturar el contenido sin procesar de la solicitud, json\_decode(..., true) convierte este contenido de formato JSON a un arreglo PHP.



Aquí vamos a validar que los campos que son requeridos est[en presentes en los datos enviados, si algunos de estos datos no existe, el sistema va a responder con un error en JSON y se detiene la ejecución con un exit.

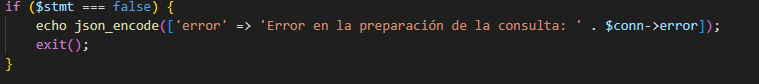
Una captura de pantalla de un celular

Descripción generada automáticamente con confianza media

Aquí los valores de los datos enviados son asignados a cada variable, ($name, $lastname, $email, $password, $id\_rol).



Tendremos una consulta SQL para insertar un nuevo registro a la tabla **User**, los signos de interrogación son marcadores de posición que serán reemplazados con los valores proporcionados para prevenir las inyecciones SQL.  
  
Osea prepare($sql) toma la consulta SQL con marcadores de posición (?) que se almacena en la variable $stmt.



if ($stmt === false) verifica si la preparaciones la consulta ha fallado, ósea si devuelve false cuando hay un error, ósea si en la sintaxis de la consulta SQL es incorrecta o si hay un problema de conexión a la base de datos. Si no fue exitosa, devuelve un mensaje en formato JSON con los detalles del error.



Bind\_param vincula los vlaores a los marcadores de posición ? en la consulta sQL , los tipos de los valores se especifican en la cadena "ssssi":

* s es para string (cadena de caracteres) y se utiliza para name, lastname, email, password.
* i es para integer (entero) y se utiliza para id\_rol.

Texto

Descripción generada automáticamente

$stmt->execute() ejecuta la consulta con los parámetros ya vinculados.

Si la ejecución es exitosa, se devuelve un mensaje de éxito.

Interfaz de usuario gráfica

Descripción generada automáticamente

Se cierra la sentencia preparada después de ejecutarla para liberar los recursos.

**My Account**

Aquí va a ser la vista de perfil de usuario en el frontend, por lo que permitirá visualizar, eidtar y guardar la información del usuario, incluyendo datos como nombre, apellido, etc.

Texto

Descripción generada automáticamente

Importamos lo necesario, lo cual hará que el proyecto funcione.

* useState y useEffect son hooks para poder uno manejar el estado y los efectos secundarios
* useNavigate, parte de react-router-dom y se usa para la navegación entre rutas
* axios, pues librería para solicitudes HTTP
* Modal y Button son componentes de boostrap que se usan para mostrar un modal y un botón

Liego definimos los componentes de estado

Texto

Descripción generada automáticamente

Profile es un objeto que contiene todos los campos del perfil del usuario, cada campo se inicializa como una cadena vacía ‘’, excepto id\_rol que se inicializa como 1, ya que id\_rol solo puede ser desde 1 hasta 3.

setProfile será la función que actualiza el estado del perfil.

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

El manejo de showModal el cual controla si el modal está visible o no, se inicia como falta, ósea no visible

editedProfile es una copia del perfil cual, pero el cual modifica en el formulario de edición.

Texto

Descripción generada automáticamente

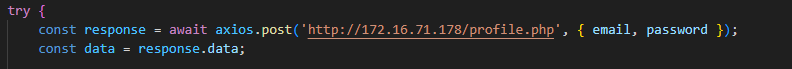
El localStorage.getItem va a recuperar el email y el password almacenados en el navegador para autenticar la solicitud, si no están, se redirigen al usuario de login.

Texto

Descripción generada automáticamente

Posteriormente tenemos el uso del hook useEffect para ejecutar una acci[on cuando se cargan los datos de email y password, fetchProfile envía la solicitud HTTP al PHP para obtener los datos del perfil del usuario y luego actualizar el estado del componente con los datos.

El useEffect permite ejecutar una función cuando el componente se monta, actualiza, desmonta, ósea que la función se ejecutará cada vez que el email o password cambien, luego fetchProfile es una función asíncrona que va a hacer solicitud HTTP al PHP para obtener los datos del perfil de usuario, la palabra **async** indica que la función contiene operaciones asíncronas, como solicitudes a servidores.



Try va intentar ejecutar el código,

Await axios.post utiliza axios para hacer una solicitud HTTP POST al profile.php, envía el email y password como datos del cuerpo de la solicitud.

* Await espera a que la solicitud se complete antes de continuar con el código
* Axios.post: hace la petición POST al servido, y envía los datos en el cuerpo de la petición.
* Const response, al completarse la solicitud, el resultado se almacena en la variable response
* Const data = response.data va a extrar los datos de la respuesta HTTP. El servido va a responder con un JSON que contiene el perfil de usuario, y se guarda en data

Texto

Descripción generada automáticamente

Acá setProfile será la función que actualiza el estado del perfil del usuario en el componente React, toma un objeto como argumento con los datos del perfil del usuario.

Luego sigue el patrón que asegura que si algún campo en los datos (data) es null, se le asignará un valor por defecto, que en este caso es ´´, esto evita que se muestren campos vació o que el componente falle si no se recibe todos los datos.



Esto actualiza el estado editedPRofile con los datos completos del perfil del usuario, para que este estado refleje el perfil actual, se usa para mantener los cambios cuando el usuario edita su perfil.

Texto

Descripción generada automáticamente con confianza media

Catch captura cualquier error que ocurra durante la ejecución del try.

Console.error imprime el error en la consola, y si hay algún error, se registrará

Texto

Descripción generada automáticamente

Luego tenemos condicional que verifica que tanto el email como el password estén definidos, si ambos existen, entonces procede a llamar la función fetchProfile. Esto asegura que no se intente obtener un perfil si las credenciales no están disponibles

Luego tenemos la parte final del useEffect, recordemos que el useEffect se utiliza para ejecutar código cuanod ocurren ciertos cambios en el componentes, este puede recibir dependencias, que son valores que React está “vigilando”. Cada vez que uno de estos valores cambine, el código dentro del useEffect se vuelve a ejecutar. Osea la primera vez que el componente se carga, useEffect se ejecuta, si el email y password tienen valores, fetchPRofile se ejecuta para obtener los datos del perfil, si el email o password cambian en algún momento mientras el componente esta montado (como iniciar sesión con otras credenciales), useEffect se vuelve a ejecutar, haciendo que React vuelva a llamar la función fetchProfile, para obtener el perfil actualizado con las nuevas credenciales.

Texto

Descripción generada automáticamente

Esta función se llama cuando el usuario hace click en “EDIT PROFILE” con el fin de cambiar el estado de showModal a true, lo que hace que el modal se muestre.

setShowModal(true) se usa el setter del estado setShowModal para indicar que el modal debe ser visible, cambiando el estado a true para abrirlo

Texto

Descripción generada automáticamente

Se ejecuta cuando el usuario quiere cerrar el modal, ósea CANCELAR. Cambia el estado de showmodal a false, para que se cierre.

Captura de pantalla de un celular

Descripción generada automáticamente

Se ejecuta cuando se da click al botón “GUARDAR CAMBIOS” dentro del modal. console.log('Datos enviados:', editedProfile) va a imprimir en la consola los datos que se enviarán al PHP, contenidos bajo editedProfile.

Por lo que se ralizara la solicitud POST al PHP, al archivo updateProfile.php, que procesa la actualización del perfil en la base de datos, editedProfile es el objeto que contiene los datos modificados por el usuario, que se envían al servidor para acutalizarlos en la base de datos.

Después de que el servidor confirme que la actualización fue exitosa, actualiza los estados de perfil con los valores editados, ósea almacenados en editedProfile. POR LO QUE INMEDIATAMENTE REFLEJA LOS CAMBIOS, luego se cierra el modal.

Texto

Descripción generada automáticamente

Esta función se ejecuta cada vez que el usuario cambia un valor en uno de los campos del formulario, por lo que primero obtiene los valores del campo,

Evento.target representa cada campo de entrada que se va a editar

Name es el nombre del campo

Value es el valor que el usuario va a ingresar en ese campo.

Luego actualiza el estado editedProfile, por lo que setEditedProfile actualiza el estado editedProfile, que contiene los datos del perfil que se están editando. prevState => ({...prevState}) copia el estado anterior del editedPRofile y [name]: value actualiza solo los campos que se están modificando, utilizando el nombre del campo como clave.

Por lo que esto asegura que el perfil editado se mantenga actualizado a medida que se cambian los valores de los campos

Pantalla de computadora con letras

Descripción generada automáticamente con confianza media

Esta parte del código verifica si el email o el password no están definidos en el localStorage, si no lo están, lo va a entender que es guest.

Acá sí email o password son null, es porque no ha iniciado sesión, !email || !password verifica que ambos valores estén presentes, si alguno falta, muestra el mensaje de redirigirlos.

Por último, si no está autenticado, muestra el mensaje de no tiene cuenta, navigate('/'): Redirige al usuario a la página de inicio de sesión (login).

Texto

Descripción generada automáticamente

Acá va a renderizar la interfal del perfil del usuario y utiliza un modal para editar los perfiles.

* onClick={handleEditClick}: Asigna la función handleEditClick para abrir el modal cuando el usuario hace clic en "Edit Profile".
* <p>{profile.name}</p>: Muestra el valor de la propiedad **name** del objeto **profile**. React reemplaza {profile.name} con el valor real que está almacenado en el estado profile.
* Este bloque se repite varias veces para mostrar otras propiedades del perfil como lastname, email, password, y telephone.

Texto

Descripción generada automáticamente

<Modal show={showModal}>, este componente de Bootstrap que muestra el modal cuanod showModal sea true.

onHide={handleClose} define lo que sucede cuando se cierra el modal. Llama a la función handleClose para ocultar el modal.

<Modal.Header> es el encabezado del modal, donde incluye un botón de cerrar y el título de Edit Profile

<Modal.Body> contiene el formulario para editar

Dentro del formulario, tenemos value={editedProfile.name || ""} el cual nos va asegurar que el campo muestre el valor actual del perfil que está siendo editado o una cadena vacía si no existe y onChange={handleInputChange} esta función se ejecutacuadno el usuario cambia algo en el formulario, actualizando el editedrofiled con los nuevos valores ingresados. Se repite este bloque para cada campo que se puede editar.

Texto

Descripción generada automáticamente con confianza media

Boton Cancel, llama la función handleClose para cerrar el modal sin guardar los cambios.

Boton Save Changes llama a handleSaveChanges para enviar los datos editar al PHP y los guarda.

PHP

Profile.php

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

file\_get\_contents("php://input") lee la solicitud http (los datos que el front envió al servidod en su solicitud POST)

json\_decode(..., true) va a convertir los datos JSON que vienen en la solicitud a un arreglo asociado al PHP

$email = $data['email'] y $password = $data['password'] asigna los valores de email y password que fueron enviados por el usuario a las variables correspondientes.

Texto

Descripción generada automáticamente

$sql contiene la consulta SQL para seleccionar los datos del perfil del usuario, busca un usuario que coincida con el email y password seleccionado.

$stmt = $conn->prepare($sql) prepara la consulta SQL para su ejecución

$stmt->bind\_param("ss", $email, $password) asocial los valores de email y password a los marcadores de la posición ? en la consulta

* Ss especifica que los parámetros son de tipo string

$stmt->execute() ejecuta la consulta

$result = $stmt->get\_result() almacena los resultados de la consulta en result

Texto

Descripción generada automáticamente

if ($result->num\_rows > 0 verifica s ila consulta devolvió algun resultado, si num\_rows es mayor a 0 significa que se encontró un usuario con email y password. Cabe recordar que esta línea verifica que si la consulta SQL devolvió al menos un resultado, ($result->num\_rows devuelve el numero de filas que tiene el resultado de la consulta

$row = $result->fetch\_assoc() Si se encontro el usuario, esta línea obtiene una fila del resultado de la consulta y la convierte a un arreglo asociatvo, la claves en este arreglo son los nombres de las colunas de la tabla (name, lastname, email) y los valores son los datos correspondientes del usuario

Después de ejecutarlo, el valor de ROW será un arreglo asociativo.

echo json\_encode($row), si el usuario es encontrado, se envía una respuesta en JSON con los datos del perfil

sino, se envia una respuesta en JSON con un mensaje de error

$stmt->close(): Cierra la sentencia preparada para liberar los recursos.

$conn->close(): Cierra la conexión a la base de datos para liberar los recursos y mantener la aplicación eficiente.

updateProdile.php

Texto

Descripción generada automáticamente

file\_get\_contents("php://input") lee el contenido de la solicitud http y json\_decode(..., true) convierte esos datos JSON en un arreglo asociativo en PHP, al ser true, JSON se convierte en un arreglo

Cada uno de los valores providnen del arreglo $data, que contiene la información enviada desde el frontend, se asigna a variables como $name, $lastname, etc, que se usarán para actualizar los datos en la base de datos

En la línea del CVV, se utiliza (int) para convertirlo a un entero, ya que los CVV son numéricos.

Texto

Descripción generada automáticamente

$sql conteine la instrucción SQL que se ejecutará para acutalizar los datos del usuario en la tabla User, y se actualiza los datos cuyo **email** coincida con el proporcionado, los ? son **marcadores de posición,** significa que se utilizará para insertar los valores de la variable de manera segura.

$stmt = $conn->prepare($sql): Prepara la consulta SQL para su ejecución.

Texto

Descripción generada automáticamente

bind\_param("sssssssssssis", ...): Vincula las variables a los marcadores de posición (?) en la consulta SQL

$stmt->execute(); ejecuta la consulta SQL con los valores proporcionados, actualizando los datos del usuario en la base de datos.

Texto

Descripción generada automáticamente

$stmt->affected\_rows: devuelve el numero de filas afectadas por la consulta SQL

Si es mayor que 0, significa que la actualización fue exitosa y el perfil del usuario fue modificado en la base de dato, sino devuelve ninguna fila, se devuelve un error

echo json\_encode(...) envía la respuesta en formato JSON al frontend, si la actualización es exitosa, envía mensaje de éxito, de lo contrario, envía un mensaje de error con los detalles de error.

$stmt->close(): Cierra la declaración preparada para liberar los recursos.

$conn->close(): Cierra la conexión a la base de datos.

**UserAdmin**

Texto

Descripción generada automáticamente

En este caso, user es un arreglo que almacena la lista de los usuarios

showModal es el que controla si el modal de agregar o editar el usuario

isEditing indica si se está editan um usuário (lanzará true) o si esta agregando uno nuevo (false)

neUser será el objeto que almacena los datos de un nuevo usuario o del usuario que se esta editando, los valores se iniciarán vacíos por defecto

Texto

Descripción generada automáticamente

fetchUsers es una función asíncrona que hace una solicitud GET al servidor para obtener la lista de los usuarios, si la respuesta es un arreglo, actualiza el estado user con los datos obtenidos

Array.isArray(response.data), esta función de JS verifica si el datos recidibo es un arreglo, si el resultado es true, significa que response.data es un arreglo, false es que no, al ser true, se llama a la función setUser para actualizar el estado del componente con los del arreglo, setUser se usa para actualizar el estado del componente.

Una captura de pantalla de un celular con texto e imagen

Descripción generada automáticamente con confianza media

Esta función se ejecuta cada vez que el usuario cambia algo en el formulario,

HandleInputChange va actualizar el estado de newUser a medida que se lleve el formulario, si el campo es id\_rol o status, lo convierte a entero, ya que son campos numéricos.

handleInputChange será la función de manejos de eventos que se activará cada vez que ocurra un cambio en los campos de formulario, la **e** representa el evento que contiene la información sobre el campo que cambió.

const { name, value } = e.target este es el elemento del formulario, name es el atributo name del atributo, value es el valor actual que el usuario ha ingresado.

Con lo que es setNewUser(prevState => ({ ...prevState, ... }))

setNewUSer será la funcón que se usará para actualizar el estado newUSer, por lo que contiene la información que se está creando o editando,

prevState va a ser el estado anterior del newUSer.

...prevState, … es un spread que copia todos los valores del estado anterior de prevState en el nuevo estado, aseguranod que los campos no modificados conserven su valor actual.

parseInt(value, 10), convierte en número entero usando parseInt, y el argumento (10) asegura que la conversión se haga a base decimal

Texto

Descripción generada automáticamente

handleShowModal se ejecuta cuando el usuario hace clic en **Add New Profile**, por lo que resetea el objeto newUser y establece el estado isEditing false, por lo que indica que se está agregando u nnuevo usuario

Texto

Descripción generada automáticamente

handleEditUSer es una función que toma como argumento el objeto user. User contiene los datos del usuario que se quiere editar. Por lo que setNewUser esta función se usa para actualizar el estado de newUser. Por lo que setNewUser(user) estamos llenando el formulario del modal con los datos del usuario seleccionado.

Captura de pantalla de un celular

Descripción generada automáticamente

handleAddUSer es una función asíncrona que se encarga de agregar un nuevo usuario.

Async indica que es asíncrona y permita usar await dentro de la función para manejar operaciones que toman tiempo como las solicitudes HTTP

Try se usa para envolver el código que podría producir errores como enviar la solicitud HTTP al servidor, si ocurre algún error durante la ejecución try, ese catch(error) se ejecuta, capturando el error y mostrando el error.

**console.log("Datos que se envían:", newUser), console.log** imprime en la consola los datos del usuario que se están enviando al servidor, almacenado en el estado newUSer.

newUser es el objeto que contiene los datos del nuevo usuario.

**const response = await axios.post('http://172.16.71.178/addProfile.php', newUser);** en esta linea Axios, es la librería para realizar solicitudes HTTP, para enviar una solicitud POST al backend,, newUser es el objeto con los datos del usuario que se enviará al servidor, los cuales fueron ingresado en el formulario por el administrador.

Await espera que la solicitud HTTP sea completada antes de continua con el resto del código, response almacenará la respuesta del servidor.

**console.log("Usuario agregado:", response.data**), después de recibir la respuesta del servidor, esta línea imprime en la consola el contenido de la respuesta, mas que todo **response.data,** como un mensaje de confirmación.

**fetchUsers();** esta función se llama después de que se haya agregado correctamente, esta función realiza una nueva solicitud al servidor para obtener la lista actualizada de usuarios, asegurándosede que el nuvo usuario aparezca en la lista.

Texto

Descripción generada automáticamente

const handleSaveChanges = () => { es la función que se ejecuta cuando el usuario hace clic en el botón “Guardar Cambios en el modal. Esta no es una función asíncrona porque solo sierve para decidir qué otra función se ejecutará

isEditing es una variable de estado que indica si el modal está en modo edición o agregar.

Una captura de pantalla de un celular con texto e imagen

Descripción generada automáticamente con confianza media

const handleDeleteUser = async (id\_user) => es una función asíncrona que recibe como argumento el id\_user,

Pantalla de un celular con letras

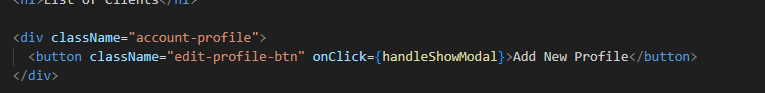
Descripción generada automáticamente

useEffect es un hook que se utiliza para ejecutar efecto secundarios, como llamadas a una API, en esta caso, en esa función llama a fechUsers() para obtener la lista de usuarios, luego hay un arreglo de dependencias que le dice a React cuándo debe volver a ejecutar el efecto.

Por lo que i [] si este arreglo esta vación, solo se ejectuará el efecto unavez, cuanod el componente se muestra en la pantalla, si esta tiene variables, se volverá a ejecutar el efecto cadavez aue esas variables cambian.

fetchUsers() es la función que se realiza una solicitud al backend para obtener la lista de usuarios, por lo que esta se ejecuta inmediatamente después de que el componente se monta, para que esté listo en la pantalla, el sistema envía la solicitud para obtener la lista de usuarios y mostrarlos.

En este caso, useEffect asegura de que la solicitud para obtener la lista de usuarios con fetchUsers y solo se ejecute una vez cuando el componente se cargue, evitando hacer múltiples solicitudes cada vez que el componente se renderiza.



Se llama la función handleShowModal, para abrir el modal que luego se puede agregar un nuevo usuario.

Tabla <table>, va a contener varias columnas, <thread> es el encabezado de cada etiqueta.

<tbody>, acá pasa lo siguiente:

Texto

Descripción generada automáticamente

**Array.isArray(user) && user.length > 0** verifica que user sea un arreglo y tiene elementos, si hay usuarios, las celdas <td> muestran las propiedades de cada usuario, por ejemplo user.name.



handleEditUser(user): Cuando se hace clic en el botón "Edit", se llama a la función handleEditUser con los datos del usuario seleccionado para editar.

handleDeleteUser(user.id\_user): Cuando se hace clic en el botón "Delete", se llama a la función handleDeleteUser para eliminar al usuario correspondiente.

Si user no es un arreglo o esta vacio, se muestra una fila con la celdas que dice No users found.

Texto

Descripción generada automáticamente

<Modal.Title>{isEditing ? 'Edit User' : 'Add New User'}</Modal.Title>

Es este caso isEditing es una variable booleana que su valor determina que texto del modal mostrara, ? es una parte de operador que pregunta "¿Es isEditing verdadero?"

'Edit User' si isEditing es true, el titulo será Edit User, : El símbolo para separar valor verdadero del valor falto,

Si isEditing es false, el titulo será Add New User

Por otra parte, utiliza la función handleInputChange para actualizar los valores conforme el usuario escribe.

**PHP**

**User.php**



$sql esta es la consulta SQL que selecciona todos los campos necesarios de la tabla User. $conn->query($sql) ejecuta la consulta SQL y guarda los resultados en result

Texto

Descripción generada automáticamente

!$result: Verifica si la consulta SQL falló. Si $result es false, significa que hubo un error al ejecutar la consulta.

$conn->error: Muestra un mensaje con el error que ocurrió durante la ejecución de la consulta.

Si hay un error, se envía un mensaje en formato JSON con la descripción del problema y se detiene la ejecución con exit().

Texto

Descripción generada automáticamente

$result->num\_rows > 0 verifica si hay filas, en el resultado de la consulta, si el numero de filas es mayor que 0, ejecuta el bloque del código dentro de este if.

$result->fetch\_assoc() acá obtiene cada fila del resultado de la consulta cmo un array asociativo (clave valor), cada campo de la base de datos es una clave, y su valor correspondiente es el valor almacenado en la base de datos.

$user[] = $row: Cada fila obtenida se agrega al array $user.

echo json\_encode($user): Convierte el array $user a formato JSON y lo envía como respuesta. Así, el frontend recibe la lista de usuarios en formato JSON.

**updateProfile2.php**

Texto

Descripción generada automáticamente  
?? null es un operador null, lo que significa que si el valor no existe, se asigna null. Esto evita errores de algún de los campos no es enviado.

isset($data['cvv']) ? (int) $data['cvv'] : null, para el cvv, se asegura de que viste y lo cinverte en un entero.

Imagen que contiene Texto

Descripción generada automáticamente

Verifica que se haya proporcionado el id\_user, si no esta presente, se envía un mensaje de error en formato JSON.

Texto

Descripción generada automáticamente

Se comienza a preparar la consulta para actualizar los campos en la tabla User, inicialmente se construye la parte Update User SET y luego se van agregando los campos que se quieren actualizar.

$sql aquí se empieza a construir la consulta SQL, actualiza los campos de la tabla user.

$updates: este array que almacenará los campos que se van a actualizar, como name = ?

$types almacena los tipos de datos de los parámetros, "s" para string y "i" para integer

$params almacena los valores correspondientes a cada parámetro que se usará em la consulta. Texto

Descripción generada automáticamente



Lo que hace es agregar el id\_user para identificar al usuario que se va a actualizar en la base de datos, usando la cláusula WHERE id\_user = ?

Ejemplo: Si el id\_user es 5, la consulta SQL incluirá WHERE id\_user = 5, lo que asegura que solo el usuario con ese id\_user será actualizado.



Combina todos los campos que se quieren actualizar en la cadena de texto separada por comas, luego agrega la condición WHERE id\_user = ? para identificar al usuario.

Si se enviaron los campos name y email, la consulta SQL final podría verse algo así

UPDATE User SET name = ?, email = ? WHERE id\_user = ?

**addProfile.php**

Texto

Descripción generada automáticamente

Extrae los valores de los campos enviado de la solicitud (nombre, apellido, email, etc.) usando el operador ??, si algún campo no está presente, se le asigna valor por defecto null.

Texto

Descripción generada automáticamente

Comprueba que los campos obligatorios (name, lastname, email, password) estén presente y o estén vacíos, si alguno falta, enviar un error en formato JSON y detiene la ejecución.

Texto

Descripción generada automáticamente

Este prepara la consulta SQL para insertar un nuevo valor en la tabla User de la base de datos, ? se usa para evitar inyecciones SQL, cada campo corresponde con un valor que será proporcionado.

**deleteProfile.php**

Texto

Descripción generada automáticamente

Verifica que el campo id\_user esté presente en la solicitud, si no se envió, se devuelve un mensaje de error. $id\_user = $data['id\_user']; Si el id\_user está presente, se asigna a la variable $id\_user.



Prepara la consula SQL que elimina el usuario de la tabla USer cuyo id\_user coincidía con el valor proporcionado, el ? utilizará para evitar inyecciones SQL y su valor será vinculado.

PRODUCTADMIN.JS

Pantalla de computadora con letras

Descripción generada automáticamente

Se hace declaración de los estados,

* Products: Almacena la lista de productos obtenida del backend
* Categories, almacena las categorías disponibles
* selectedCategories, almacena las categorías seleecionadas para el producto actual
* showModal controla el modal
* isEditing, indica si estamos agregando un nuevo productos o editando
* currentProductID, almacena el ID del producto que está siendo editando
* newProduct, conteine los datos del productos que se está creando o editando

Texto

Descripción generada automáticamente

Para poder realizar la solicitud al servidor se usa axios.get, cn el fin de obtener los datos de los productos, si la respuesta es un array, actualiza el estado products con los datos obtenidos.

Se declara la función fetchProducts, async indica que la función es asincrónica lo que significa que puede realizar varias operaciones que toman tiempo, como solicitudes a servidores sin bloquear la ejecución del resto del código.

Luego se usa axios.get para hacer una solicitud HTTP de tipo GET al servidor, la repuesta del servidor lo devuelve product.php y se almacena en la variable response.

console.log("API Response:", response.data) imprime en la consola el contenido de response.data, que es donde Axios guarda los datos de la respuesta.

Response.data es la propiedad donde se almacena los datos devueltos por el servidor. Es un array de usuarios.

if (Array.isArray(response.data)) verifica que los datos devueltos por el servidor son un array, si es un array, significa que la respuesta es válida y que contiene los usuarios.

setProducts(response.data) si la respuesta es valida un array, llama a setProducts para actualizar el estado user con los datos recibidos.

Pantalla de computadora con letras

Descripción generada automáticamente con confianza media

const fetchCategories = async () => acá se declara una función asincrónica fetchCategories. Async permite la función ejecute las operaciones asincrónicas y espere su finalización antes de continuar

axios.get realiza la solicitud HTTP de tipo GET para el HTTP, este HTTP está en un servidor y debería devolver una lista de categorías en formato JSON.

Response almacena la repuesta del servidor, los datos devueltos por el servidor estarán en la propiedad response.data

if (Array.isArray(response.data)) verifica que los datos que devuelve el servidor son un array, la función Array.isArray() se usa para verificar que el valor que hemos recibido si sea un array,

setCategories(response.data.map(cat => ({ value: cat.id\_category, label: cat.name })));

response.data.map() itera sobre cada elemento del array response.data donde cada elemento representa una categoría. Para cada categoría se crea un nuevo objeto con las propiedades.

* **value**: Se establece como el valor del campo id\_category de la categoría (es el identificador único de la categoría).
* **label**: Se establece como el valor del campo name de la categoría (el nombre de la categoría que se mostrará).

Este nuevo array de objetos se pasa a la función setCategories, que es la función para actualizar el estado categories.

Texto

Descripción generada automáticamente

const handleInputChange = (e) => declara la función llamada handleInputChange, donde e es el evento que se desencadena cuando un usuario interactúa con los campos, este evento contiene información como el nombre del campo y su valor actual.

const { name, value } = e.target extrae las propiedades name y value del campo de entrada que dispara el evento

* e.target hace referencia al campo que generó el evento
* name es el atributo que identifica el campo, como name Price, inventory
* value es el valor que el usuario ingresó en ese campo de input

setNewProduct(prevState => acá setNewProduct es la funcón que actualiza el estado newproduct

prevState es el estado anterior de newProduct antes de realizar cualquier cambio, react nos permite acceder a ese estado anterior para poder hacer cambios de manera más controlada.

const updatedProduct = {

...prevState,

[name]: value

Usa … para copier todas las propiedades del esado anterior (prevState) a un nuevo objeto llamado updateProduct

Luego, se agrega a una propiedad o actualiza una existente, basada en el campo de entrada.

[name] usa el valor del campo name como clave, y se le asigna el valor que el usuario ingresó en el campo (value). Así, se actualiza dinámicamente la propiedad correcta en newProduct.

**if (name === 'inventory') {**

**updatedProduct.stock = parseInt(value) === 0 ? '0' : '1';**

acá si el nombre del campo modificado es inventory, entonces se ejecuta una lógica.

Si el nombre del campo modificado es 'inventory', entonces se ejecuta una lógica adicional.

Convierte el valor del inventario a un número entero usando parseInt(value).

Si el inventario es 0 (lo que indica que no hay productos en stock), el campo stock se actualiza a '0' (lo que significa que el producto está "fuera de stock").

Si el inventario es mayor que 0, entonces el campo stock se actualiza a '1' (lo que indica que el producto está "en stock").

Esta lógica es útil para mantener la coherencia entre el inventario y el estado de stock de un producto.

**return updatedProduct;**

Retorna el objeto updatedProduct que contiene el nuevo estado del producto con las actualizaciones realizadas.

React usa este valor retornado para actualizar el estado de newProduct.

 **Escucha los cambios** que hace el usuario en los campos de un formulario.

 **Extrae** el nombre del campo (name) y el valor ingresado (value).

 **Actualiza** el estado del producto (newProduct) con el nuevo valor del campo modificado.

 **Si el campo modificado es el inventario**, actualiza automáticamente el estado de stock (stock) basado en el valor del inventario.

 Finalmente, retorna el nuevo estado actualizado para reflejar los cambios en la interfaz de usuario.

Texto

Descripción generada automáticamente

* **const handleCategoryChange = (selected) => {**:
  + Declara una función llamada **handleCategoryChange** que toma un parámetro **selected**.
  + **selected** es el valor que se recibe cuando el usuario selecciona una o varias categorías en el formulario. Este parámetro es un array de objetos que representan las categorías seleccionadas.
* **setSelectedCategories(selected.map(item => item.value));**:
  + Utiliza **setSelectedCategories** para actualizar el estado **selectedCategories** con un array de valores de las categorías seleccionadas.
  + **selected.map(item => item.value)**: Recorre el array **selected** y extrae solo el valor (value) de cada objeto de categoría seleccionada.
  + **item.value**: Se refiere a la propiedad value de cada objeto en el array selected. value es probablemente el id\_category de cada categoría.

**Resumen de handleCategoryChange:**

* Cuando el usuario selecciona una o varias categorías, esta función actualiza el estado **selectedCategories** para que solo contenga los **id\_category** de las categorías seleccionadas. Esto permite que la aplicación maneje correctamente las categorías asignadas a un producto.

**Ejemplo:**

* Si **selected** es un array como este:

json

Copy code

[

{ "value": 1, "label": "Comida" },

{ "value": 2, "label": "Juguetes" }

]

El **map** devolvería un array de solo los valores:

json

Copy code

[1, 2]

Esto representa los **id\_category** de las categorías seleccionadas.

**Resultado:**

* **setSelectedCategories(...)** actualiza el estado **selectedCategories** con este nuevo array de valores (o **id**s de categorías). De esta manera, el sistema sabe qué categorías han sido seleccionadas para el producto actual.

Texto

Descripción generada automáticamente

* **setNewProduct({...})**:
  + Establece el estado **newProduct** con valores vacíos o predeterminados para un nuevo producto.
  + Este objeto contiene las propiedades del producto que serán completadas por el usuario:
    - **name**: Cadena vacía, el nombre del nuevo producto.
    - **description**: Cadena vacía, la descripción del nuevo producto.
    - **price**: Cadena vacía, el precio del nuevo producto.
    - **inventory**: Cadena vacía, la cantidad de inventario disponible.
    - **stock**: Por defecto, es '1', lo que significa que el producto está **"In Stock"** (en inventario).
    - **comment**: Cadena vacía, cualquier comentario adicional sobre el producto.
    - **color**: Cadena vacía, el color del producto.
    - **size**: Cadena vacía, el tamaño del producto.
  + Al vaciar estos campos, el formulario en el modal estará listo para recibir los datos de un nuevo producto.
* **setSelectedCategories([])**:
  + Limpia la lista de categorías seleccionadas, dejándola vacía.
  + Esto asegura que al abrir el modal para agregar un producto, no haya categorías preseleccionadas.
* **setIsEditing(false)**:
  + Establece **isEditing** en **false**, lo que indica que el usuario **no** está editando un producto existente, sino que está agregando uno nuevo.
  + Esta bandera probablemente controla el comportamiento del modal (si se está agregando o editando).
* **setShowModal(true)**:
  + Abre el modal estableciendo **showModal** en **true**. Esto activa el componente modal en la interfaz para que el usuario pueda comenzar a completar el formulario.

**Resumen de handleShowModal:**

* Esta función prepara el estado para agregar un nuevo producto:
  + Resetea los valores del producto (nombre, precio, inventario, etc.).
  + Limpia las categorías seleccionadas.
  + Establece que estamos agregando un nuevo producto (no editando).
  + Finalmente, muestra el modal en pantalla para que el usuario pueda interactuar con él y completar el formulario.

**En conjunto:**

Estas funciones trabajan juntas para manejar la selección de categorías y la apertura del modal al agregar un nuevo producto. **handleCategoryChange** se encarga de gestionar las categorías seleccionadas, mientras que **handleShowModal** prepara el modal para la creación de un nuevo producto, asegurándose de que todos los campos estén vacíos y listos para el usuario.

Texto

Descripción generada automáticamente

**const handleEditProduct = (product) => {**

* **Lo que hace**:
  + Declara una función de flecha llamada **handleEditProduct** que toma un parámetro **product**.
  + El parámetro **product** contiene toda la información del producto que será editado (como el nombre, descripción, inventario, categorías, etc.).

**setNewProduct({ ... })**

* **Lo que hace**:
  + Usa la función **setNewProduct** (de React) para actualizar el estado de **newProduct**, que contiene los datos del producto que se están editando.
  + **newProduct** es el objeto que será usado para llenar el formulario en el modal de edición de productos.

**Desglose del objeto pasado a setNewProduct:**

* **name: product.name**: Establece el nombre del producto en el formulario basado en el nombre actual del producto.
* **description: product.description**: Establece la descripción del producto en el formulario basado en la descripción actual del producto.
* **price: product.price**: Establece el precio del producto en el formulario.
* **inventory: product.inventory**: Establece el valor del inventario del producto en el formulario.
* **stock: product.inventory === 0 ? '0' : '1'**: Evalúa el inventario del producto:
  + Si **product.inventory** es 0, significa que el producto está **fuera de stock** y por lo tanto asigna '0' al campo de **stock**.
  + Si el inventario es mayor que 0, significa que el producto está **en stock** y se asigna '1'.
* **comment: product.comment**: Establece el comentario del producto en el formulario.
* **color: product.color**: Establece el color del producto en el formulario.
* **size: product.size**: Establece el tamaño del producto en el formulario.

**setCurrentProductId(product.id\_product);**

* **Lo que hace**:
  + Usa la función **setCurrentProductId** para actualizar el estado **currentProductId** con el **ID del producto** que se está editando.
  + Este ID es importante porque cuando se guarden los cambios, el sistema sabrá qué producto actualizar en la base de datos.

**setSelectedCategories(product.categories || []);**

* **Lo que hace**:
  + Usa la función **setSelectedCategories** para actualizar el estado **selectedCategories**, que contiene las categorías seleccionadas para este producto.
  + **product.categories** se refiere a las categorías actuales del producto que se está editando. Si no hay categorías, asigna un array vacío ([]).
  + Esto se usa para que el modal de edición de productos muestre las categorías que ya están asignadas al producto actual.

**setIsEditing(true);**

* **Lo que hace**:
  + Usa la función **setIsEditing** para cambiar el estado de **isEditing** a **true**.
  + Esto es importante porque el estado **isEditing** probablemente se utiliza para distinguir entre:
    - Si el usuario está **agregando** un producto nuevo (cuando isEditing es false).
    - O si está **editando** un producto existente (cuando isEditing es true).

**6. setShowModal(true);**

* **Lo que hace**:
  + Usa la función **setShowModal** para abrir el modal de edición de productos. Al establecer **showModal** en **true**, el modal aparecerá en la pantalla y el usuario podrá comenzar a editar los campos del producto.

**Resumen General:**

La función **handleEditProduct** realiza lo siguiente cuando el usuario selecciona un producto para editar:

1. **Cargar los datos del producto** (nombre, precio, inventario, comentarios, etc.) en el formulario del modal.
2. **Asignar el ID del producto** que se está editando para poder hacer referencia a él.
3. **Cargar las categorías actuales** del producto en el selector de categorías.
4. Indicar que el **modal está en modo edición** (no en modo de agregar un nuevo producto).
5. **Abrir el modal** para que el usuario pueda ver los campos del producto y comenzar a editarlos.

Esta función es esencial para preparar el modal de edición de productos y garantizar que los datos correctos se muestren en el formulario.

Texto

Descripción generada automáticamente

**const handleInputChange = (e) => {**

* **Lo que hace**:
  + Declara una función de flecha llamada **handleInputChange** que toma un parámetro **e**, que es el evento que ocurre cuando el usuario interactúa con un campo de entrada (input).

**2. const { name, value } = e.target;**

* **Lo que hace**:
  + Extrae dos propiedades del elemento que generó el evento **e.target**:
    - **name**: Es el nombre del campo de entrada (como 'name', 'price', 'inventory', etc.). Este nombre coincide con las claves del objeto **newProduct** en el estado.
    - **value**: Es el valor que el usuario ingresó en ese campo.
  + Esta extracción facilita el uso de los valores para actualizar dinámicamente el estado del producto según el campo modificado.

**3. setNewProduct(prevState => {**

* **Lo que hace**:
  + Llama a la función **setNewProduct**, que actualiza el estado de **newProduct**.
  + **prevState** es el estado anterior de **newProduct**, que contiene los valores actuales de los campos del producto. Usar **prevState** garantiza que los otros valores en el estado del producto no se pierdan cuando actualices uno solo de los campos.

**4. const updatedProduct = { ...prevState, [name]: value };**

* **Lo que hace**:
  + Crea un nuevo objeto llamado **updatedProduct** que copia todas las propiedades del estado anterior **prevState** (utilizando el operador **spread** ...).
  + **[name]: value**: Luego, actualiza el campo específico que fue modificado por el usuario.
    - **name** es el nombre del campo de entrada y **value** es el valor ingresado por el usuario.
    - Al usar corchetes ([name]), estás asignando el valor dinámicamente al campo correspondiente de **newProduct**.

**5. if (name === 'inventory') {**

* **Lo que hace**:
  + Verifica si el campo modificado es **inventory** (inventario).
  + Si el usuario cambió el valor del inventario, se ejecuta una lógica especial para actualizar el campo **stock** del producto.

**6. updatedProduct.stock = parseInt(value) === 0 ? '0' : '1';**

* **Lo que hace**:
  + Convierte el valor del inventario en un número entero usando **parseInt(value)**.
  + Si el inventario es igual a **0**, el campo **stock** se establece en **'0'** (lo que indica que el producto está **"out of stock"** o **fuera de inventario**).
  + Si el inventario es mayor que **0**, el campo **stock** se establece en **'1'** (lo que indica que el producto está **"in stock"** o **en inventario**).
  + Esta lógica asegura que el estado de stock del producto se actualice automáticamente cuando el usuario cambie el inventario.

**7. return updatedProduct;**

* **Lo que hace**:
  + Retorna el objeto **updatedProduct**, que contiene el nuevo estado del producto con todos los cambios aplicados (el campo modificado y, si es necesario, el estado de stock).
  + React usa este valor para actualizar el estado de **newProduct**.

**Resumen General:**

La función **handleInputChange**:

1. **Detecta cambios** en cualquier campo de entrada del formulario (nombre, precio, inventario, etc.).
2. **Actualiza** el estado del producto **newProduct** con el nuevo valor ingresado por el usuario.
3. Si el campo modificado es el **inventario**, automáticamente ajusta el campo **stock** para reflejar si el producto está **en stock** o **fuera de stock**, dependiendo de si el inventario es 0 o mayor.
4. Retorna el nuevo estado actualizado para que la aplicación refleje los cambios de forma inmediata.

Este patrón es muy útil en formularios, ya que permite que los cambios en los campos se gestionen de manera eficiente y dinámica en la interfaz de usuario.

Texto

Descripción generada automáticamente

**const handleUpdateProduct = async () => {**

* **Lo que hace**:
  + Declara una función de flecha **asíncrona** llamada **handleUpdateProduct**. El uso de **async** permite el uso de **await** dentro de la función, lo que significa que la ejecución de ciertas partes del código puede esperar a que otras partes terminen antes de continuar.
  + Esta función no recibe ningún parámetro porque usa los datos almacenados en el estado.

**2. try {**

* **Lo que hace**:
  + Se inicia un bloque try...catch para manejar posibles errores al intentar actualizar el producto. Si algo falla dentro del bloque try, el flujo de control pasa automáticamente al bloque catch para manejar el error.

**3. await axios.post('http://172.16.71.178/updateProduct.php', {**

* **Lo que hace**:
  + Realiza una solicitud HTTP POST a la URL 'http://172.16.71.178/updateProduct.php' utilizando **axios.post**.
  + **await**: Pausa la ejecución hasta que la solicitud al servidor se complete (es decir, espera la respuesta del servidor).
  + La función está enviando los datos del producto al archivo **updateProduct.php** en el backend, que se encargará de actualizar el producto en la base de datos.

**Datos enviados en la solicitud POST:**

El segundo parámetro de **axios.post** es un objeto con los datos que se envían al servidor:

js

Copy code

{

id\_product: currentProductId,

...newProduct,

categories: selectedCategories

}

* **id\_product: currentProductId**:
  + Envía el ID del producto que está siendo editado. **currentProductId** es el estado que contiene el **ID** del producto actual, el cual fue establecido previamente cuando se seleccionó el producto a editar.
* **...newProduct**:
  + Usa el **operador spread** (...) para incluir todas las propiedades del estado **newProduct** (nombre, descripción, precio, inventario, etc.) en el objeto enviado al servidor.
  + Esto incluye todos los detalles del producto editado por el usuario.
* **categories: selectedCategories**:
  + También envía la lista de categorías seleccionadas (almacenadas en **selectedCategories**), que fueron seleccionadas por el usuario para este producto.
  + Esta lista se incluirá como parte de la información del producto a actualizar en el servidor.

**4. fetchProducts();**

* **Lo que hace**:
  + Llama a la función **fetchProducts** para volver a cargar la lista de productos desde el servidor.
  + **fetchProducts** obtiene la lista actualizada de productos después de que el producto ha sido modificado y la lista se vuelve a renderizar en la interfaz de usuario para reflejar los cambios.

**5. setShowModal(false);**

* **Lo que hace**:
  + Usa la función **setShowModal** para cerrar el modal de edición de productos.
  + Establecer **showModal** en **false** significa que el modal ya no se mostrará en la interfaz de usuario después de que los cambios se hayan guardado.

**6. } catch (error) {**

* **Lo que hace**:
  + Si ocurre un error durante el proceso (por ejemplo, si la solicitud POST falla), el flujo de ejecución pasa al bloque **catch**.
  + El error se captura en la variable **error** y se maneja de la siguiente manera:

**7. console.error("Error updating product:", error);**

* **Lo que hace**:
  + Usa **console.error** para mostrar un mensaje de error en la consola del navegador, junto con los detalles del error.
  + Esto es útil para la depuración en caso de que la actualización del producto falle.

**Resumen General:**

1. La función **handleUpdateProduct** envía una solicitud HTTP POST al servidor para actualizar los detalles de un producto.
2. Incluye el **ID del producto**, los **datos del producto** (como nombre, precio, inventario, etc.), y las **categorías seleccionadas**.
3. Después de una actualización exitosa:
   * Se vuelve a cargar la lista de productos llamando a **fetchProducts**.
   * Se cierra el modal de edición llamando a **setShowModal(false)**.
4. Si ocurre algún error durante el proceso, el error se captura y se imprime en la consola para su revisión.

Esta función es esencial para permitir a los usuarios editar productos en la aplicación, actualizar la base de datos en el backend y refrescar la interfaz de usuario con los nuevos datos del producto.Principio del formulario

Final del formulario

Texto

Descripción generada automáticamente

**const handleSaveChanges = () => {**

* **Lo que hace**:
  + Declara una función de flecha llamada **handleSaveChanges**. Esta función no toma parámetros porque usa el estado local de la aplicación para determinar qué hacer.

**2. if (isEditing) {**

* **Lo que hace**:
  + Verifica el valor de la variable **isEditing**.
  + **isEditing** es una variable de estado que indica si el usuario está editando un producto existente o agregando uno nuevo.
  + Si **isEditing** es **true**, significa que el usuario está en modo de edición (editando un producto ya existente).

**3. handleUpdateProduct();**

* **Lo que hace**:
  + Si **isEditing** es **true**, se llama a la función **handleUpdateProduct**.
  + **handleUpdateProduct** se encarga de enviar una solicitud al servidor para **actualizar** el producto existente. Esto incluye enviar los datos modificados y el ID del producto que se está editando.
  + Esta función ya fue explicada antes y maneja la lógica para actualizar un producto existente.

**4. } else {**

* **Lo que hace**:
  + Si **isEditing** es **false**, entonces el usuario no está editando un producto, sino que está agregando uno nuevo.
  + En este caso, se ejecuta el bloque de código en **else**.

**5. handleAddProduct();**

* **Lo que hace**:
  + Llama a la función **handleAddProduct** si **isEditing** es **false**.
  + **handleAddProduct** se encarga de enviar una solicitud al servidor para **agregar** un nuevo producto con los datos que el usuario ha ingresado en el formulario.
  + Esta función agrega un nuevo producto a la base de datos y luego actualiza la lista de productos en la interfaz de usuario.

**Resumen General:**

1. **handleSaveChanges** es una función de "decisión".
2. Si el estado **isEditing** es **true** (el usuario está editando un producto existente), se llama a la función **handleUpdateProduct** para **actualizar** el producto.
3. Si **isEditing** es **false** (el usuario está agregando un nuevo producto), se llama a la función **handleAddProduct** para **agregar** el producto a la base de datos.
4. Esta función centraliza el proceso de decisión entre las acciones de agregar o actualizar, simplificando la lógica del código y evitando duplicación.

Texto

Descripción generada automáticamente

**1. const handleDeleteProduct = async (id\_product) => {**

* **Lo que hace**:
  + Declara una función de flecha llamada **handleDeleteProduct**.
  + Esta función es **asíncrona** debido al uso de **async**, lo que permite el uso de **await** para manejar promesas de forma secuencial.
  + Recibe un parámetro **id\_product**, que es el ID del producto que se desea eliminar.

**2. if (window.confirm("Are you sure you want to delete this product?")) {**

* **Lo que hace**:
  + Muestra un cuadro de diálogo de confirmación al usuario utilizando la función **window.confirm()** del navegador.
  + Si el usuario confirma (hace clic en "OK"), se ejecuta el código dentro del bloque **if**.
  + Si el usuario cancela, no se ejecuta el código y el producto no se elimina.

**3. try {**

* **Lo que hace**:
  + Inicia un bloque **try...catch** para intentar eliminar el producto. Si ocurre un error durante el proceso, el flujo de control pasa al bloque **catch** para manejar el error.

**4. await axios.post('http://172.16.71.178/deleteProduct.php', { id\_product });**

* **Lo que hace**:
  + Envía una solicitud **HTTP POST** al archivo **deleteProduct.php** en el servidor, utilizando **axios.post**.
  + **await**: Espera a que la solicitud al servidor se complete antes de continuar con la ejecución del código.
  + **{ id\_product }**: Envía el **ID del producto** como un objeto JSON al servidor. Este ID será utilizado por el servidor para localizar y eliminar el producto correspondiente de la base de datos.

**5. fetchProducts();**

* **Lo que hace**:
  + Llama a la función **fetchProducts** para volver a cargar la lista de productos después de eliminar el producto.
  + Esto garantiza que la interfaz de usuario se actualice y muestre la lista actualizada sin el producto eliminado.

**6. } catch (error) {**

* **Lo que hace**:
  + Si ocurre un error durante el proceso de eliminación, el flujo de control pasa a este bloque **catch** para manejar el error.

**7. console.error("Error deleting product:", error);**

* **Lo que hace**:
  + Imprime un mensaje de error en la consola del navegador con los detalles del error.
  + Esto es útil para depurar problemas si la eliminación del producto falla.

**Resumen General:**

1. **handleDeleteProduct** es una función asíncrona que se utiliza para eliminar un producto de la base de datos.
2. Muestra un cuadro de confirmación al usuario antes de eliminar el producto.
3. Si el usuario confirma la acción, envía una solicitud POST al servidor con el **ID del producto** para eliminarlo.
4. Después de eliminar el producto, se vuelve a cargar la lista de productos llamando a **fetchProducts** para reflejar el cambio en la interfaz.
5. Si ocurre un error, el mensaje de error se imprime en la consola para ayudar en la depuración.

Esta función es crucial para permitir que los administradores eliminen productos de la aplicación y reflejen el cambio en tiempo real en la interfaz de usuario.

Texto

Descripción generada automáticamente

**useEffect(() => {**

* **Lo que hace**:
  + Define un **hook de efecto** (**useEffect**), que es una función de React que se ejecuta después de que el componente ha sido renderizado (montado).
  + Este **hook** se utiliza principalmente para ejecutar código que necesita ejecutarse como efectos secundarios (como llamadas a APIs o suscripciones) cuando se monta o actualiza el componente.

**2. fetchProducts();**

* **Lo que hace**:
  + Llama a la función **fetchProducts**, que realiza una solicitud al servidor para obtener la lista de productos disponibles y los almacena en el estado local del componente.
  + **fetchProducts** se ejecuta cuando el componente se monta por primera vez para mostrar los productos disponibles en la interfaz de usuario.

**3. fetchCategories(); // Fetch available categories (tags)**

* **Lo que hace**:
  + Llama a la función **fetchCategories**, que realiza una solicitud al servidor para obtener la lista de categorías (o tags) disponibles.
  + **fetchCategories** es esencial para mostrar las opciones de categorías al usuario cuando agrega o edita un producto.
  + También almacena estas categorías en el estado para que estén disponibles en el componente de selección.

**4. }, []);**

* **Lo que hace**:
  + Este array vacío **[]** es un segundo argumento del **useEffect**, conocido como la **lista de dependencias**.
  + **[]** indica que este **useEffect** solo se ejecutará **una vez**, cuando el componente se monte por primera vez.
  + Como el array está vacío, significa que las funciones **fetchProducts** y **fetchCategories** no se volverán a ejecutar a menos que el componente se desmonte y vuelva a montarse.

Texto

Descripción generada automáticamente

**<thead> y <tr>**

* **Lo que hace**:
  + **<thead>**: Define el encabezado de la tabla.
  + **<tr>**: Representa una fila en la tabla. Dentro de esta fila, hay varios **<th>** (table headers), cada uno representando un encabezado para las columnas de **ID**, **Name**, **Description**, **Price**, **Inventory**, **Categories**, y **Options**.

**3. <tbody>**

* **Lo que hace**:
  + Contiene el cuerpo de la tabla, donde se mostrarán los productos.

**4. {Array.isArray(products) && products.length > 0 ? ( ... ) : ( ... )}**

* **Lo que hace**:
  + Este bloque es una condición que verifica si **products** es un array válido y si contiene al menos un producto.
  + Si la condición es **true**, la aplicación mapeará los productos en la tabla.
  + Si no hay productos (la condición es **false**), mostrará un mensaje: **"No products found"**.

**5. products.map(product => ( ... ))**

* **Lo que hace**:
  + Utiliza la función **map()** para recorrer la lista de productos, y por cada producto, renderiza una fila **<tr>** con los datos correspondientes.

**6. <tr key={product.id\_product}>**

* **Lo que hace**:
  + Cada producto genera una fila en la tabla (**<tr>**) con un atributo **key** que es único, en este caso, **id\_product**. Esto ayuda a React a identificar cada fila de manera única.

**7. <td>{product.id\_product}</td> a <td>{product.inventory}</td>**

* **Lo que hace**:
  + Renderiza varias celdas **<td>** que contienen los datos del producto: ID, nombre, descripción, precio, inventario.

Texto

Descripción generada automáticamente**4. {Array.isArray(products) && products.length > 0 ? ( ... ) : ( ... )}**

* **Lo que hace**:
  + Este bloque es una condición que verifica si **products** es un array válido y si contiene al menos un producto.
  + Si la condición es **true**, la aplicación mapeará los productos en la tabla.
  + Si no hay productos (la condición es **false**), mostrará un mensaje: **"No products found"**.

**5. products.map(product => ( ... ))**

* **Lo que hace**:
  + Utiliza la función **map()** para recorrer la lista de productos, y por cada producto, renderiza una fila **<tr>** con los datos correspondientes.

**6. <tr key={product.id\_product}>**

* **Lo que hace**:
  + Cada producto genera una fila en la tabla (**<tr>**) con un atributo **key** que es único, en este caso, **id\_product**. Esto ayuda a React a identificar cada fila de manera única.

**7. <td>{product.id\_product}</td> a <td>{product.inventory}</td>**

* **Lo que hace**:
  + Renderiza varias celdas **<td>** que contienen los datos del producto: ID, nombre, descripción, precio, inventario.

**8. <td>{Array.isArray(product.categories) ? product.categories.join(', ') : 'No categories'}</td>**

* **Lo que hace**:
  + Verifica si **product.categories** es un array. Si lo es, combina las categorías en una sola cadena separada por comas.
  + Si no hay categorías, muestra el mensaje **No categories**.

**9. <td> con los botones "Edit" y "Delete"**

* **Lo que hace**:
  + Renderiza un botón para **Editar** y uno para **Eliminar** el producto.
  + El botón **Editar** llama a la función **handleEditProduct(product)** con los datos del producto seleccionado.
  + El botón **Eliminar** llama a **handleDeleteProduct(product.id\_product)** con el ID del producto a eliminar.

**10. {product.categories || []}**

* **Lo que hace**:
  + Verifica si el producto tiene categorías. Si **product.categories** es **null** o no está definido, se utiliza un array vacío como valor por defecto.

Texto

Descripción generada automáticamente

**<Modal.Title>{isEditing ? 'Edit Product' : 'Add New Product'}</Modal.Title>**

* **Lo que hace**:
  + Cambia dinámicamente el título del modal según si el usuario está editando un producto o agregando uno nuevo.
  + Si **isEditing** es **true**, el título será **Edit Product**.
  + Si **isEditing** es **false**, el título será **Add New Product**.

Texto

Descripción generada automáticamente**<Select ... /> (react-select)**

* **Lo que hace**:
  + Un componente de selección múltiple para las categorías del producto.
  + **options={categories}**: Define las opciones disponibles basadas en las categorías obtenidas del servidor.
  + **isMulti**: Permite seleccionar múltiples categorías.
  + **onChange={handleCategoryChange}**: Actualiza las categorías seleccionadas cuando el usuario cambia las opciones.

Category.php

Texto

Descripción generada automáticamente

**$sql = "SELECT c.id\_category, c.name, GROUP\_CONCAT(p.name) AS parent\_names ..."**

* **Lo que hace**:
  + Esta línea define la consulta SQL que se utilizará para obtener las categorías.
  + **c.id\_category**: Selecciona el ID de la categoría principal.
  + **c.name**: Selecciona el nombre de la categoría principal.
  + **GROUP\_CONCAT(p.name)**: Utiliza la función **GROUP\_CONCAT** para concatenar los nombres de las categorías padres (si existen). Si una categoría tiene múltiples padres, estos se concatenarán en una sola cadena separada por comas.
  + **AS parent\_names**: Le asigna un alias a esta concatenación, para que pueda ser referida como **parent\_names** en el código.

**2. FROM Category c**

* **Lo que hace**:
  + Define la tabla **Category** con el alias **c** como la tabla principal de la consulta. Esta tabla contiene las categorías.

**3. LEFT JOIN CategoryParent cp ON c.id\_category = cp.id\_category**

* **Lo que hace**:
  + Utiliza un **LEFT JOIN** con la tabla **CategoryParent**, que es la tabla intermedia que maneja las relaciones de padre-hijo entre categorías.
  + **c.id\_category = cp.id\_category**: Junta las categorías principales con sus respectivas categorías padres usando el campo **id\_category**.

**4. LEFT JOIN Category p ON cp.id\_parent = p.id\_category**

* **Lo que hace**:
  + Realiza otro **LEFT JOIN** con la tabla **Category** (ahora con alias **p**, que representa las categorías padres).
  + **cp.id\_parent = p.id\_category**: Junta las categorías hijas con sus respectivas categorías padres, usando el campo **id\_parent**.

**5. GROUP BY c.id\_category**

* **Lo que hace**:
  + Agrupa los resultados por el **ID de la categoría principal** (**c.id\_category**). Esto es necesario para utilizar **GROUP\_CONCAT** y concatenar los nombres de los padres correspondientes.

**6. $result = $conn->query($sql);**

* **Lo que hace**:
  + Ejecuta la consulta SQL definida en **$sql** utilizando la conexión a la base de datos (**$conn**).
  + El resultado de la consulta se almacena en la variable **$result**.

**7. if ($result->num\_rows > 0) {**

* **Lo que hace**:
  + Verifica si la consulta ha devuelto algún resultado.
  + **$result->num\_rows > 0**: Comprueba si hay más de 0 filas en el resultado.

**8. while ($row = $result->fetch\_assoc()) {**

* **Lo que hace**:
  + Utiliza un **bucle while** para recorrer todas las filas obtenidas de la consulta.
  + **$result->fetch\_assoc()**: Obtiene cada fila como un array asociativo donde los nombres de las columnas son las claves del array.

**9. $categories[] = [**

* **Lo que hace**:
  + Crea un array llamado **$categories** y, dentro del bucle **while**, va agregando cada fila de resultados (cada categoría) a este array.
  + Cada elemento en el array **$categories** contiene:
    - **id\_category**: El ID de la categoría.
    - **name**: El nombre de la categoría.
    - **parent\_names**: Los nombres de las categorías padres concatenados (o vacío si no tiene padres).

**10. echo json\_encode($categories);**

* **Lo que hace**:
  + Convierte el array **$categories** a formato JSON utilizando **json\_encode** y lo imprime (lo envía como respuesta).
  + Este JSON será devuelto al frontend, que podrá usarlo para mostrar las categorías en la interfaz.

**11. $conn->close();**

* **Lo que hace**:
  + Cierra la conexión con la base de datos después de ejecutar la consulta.

**Resumen General:**

1. El código construye una consulta SQL para obtener todas las categorías de la tabla **Category** junto con sus categorías padres (si existen), utilizando un **LEFT JOIN**.
2. Usa **GROUP\_CONCAT** para concatenar los nombres de los padres en un solo campo.
3. Si hay resultados, recorre todas las filas y las guarda en un array **$categories**.
4. Finalmente, convierte este array en formato JSON y lo envía al frontend.

Este código es útil para manejar estructuras jerárquicas de categorías, donde una categoría puede tener múltiples padres, y devuelve esta relación de manera legible para su uso en el frontend.

Up

Footer

Texto

Descripción generada automáticamente

Texto

Descripción generada automáticamente

**const userRole = localStorage.getItem('userRole');**

* **Lo que hace**:
  + Obtiene el valor de **userRole** almacenado en **localStorage**.
  + **localStorage** es un almacenamiento local del navegador donde se guardan datos de sesión del usuario. En este caso, **userRole** indica el rol del usuario, que podría ser, por ejemplo, **admin**, **employee**, o **guest**.

**7. const email = localStorage.getItem('email');**

* **Lo que hace**:
  + Obtiene el valor del **email** del usuario que está almacenado en **localStorage**.
  + El **email** será utilizado más adelante para actualizar el campo **last\_login** cuando el usuario cierre sesión.

**8. const handleLogout = async () => {**

* **Lo que hace**:
  + Define una función **asíncrona** para manejar el proceso de logout del usuario.

**8.1. if (email) { ... }**

* **Lo que hace**:
  + Verifica si el **email** existe en el **localStorage**.
  + Si existe, procede a hacer una petición **POST** a **http://172.16.71.178/lastlogin.php** para actualizar el campo **last\_login** de ese usuario en la base de datos.

**8.2. await axios.post('http://172.16.71.178/lastlogin.php', { email });**

* **Lo que hace**:
  + Realiza una solicitud **POST** al servidor utilizando **axios**. Envía el **email** del usuario para que el servidor actualice el campo **last\_login**.
  + El endpoint **lastlogin.php** es responsable de manejar esta actualización en la base de datos.

**8.3. localStorage.clear();**

* **Lo que hace**:
  + Limpia todo el contenido de **localStorage**, es decir, elimina todas las claves y valores almacenados, incluyendo **email** y **userRole**.
  + Esto se hace para "cerrar" la sesión del usuario, eliminando toda su información.

**8.4. navigate('/');**

* **Lo que hace**:
  + Redirige al usuario a la página principal de la aplicación (ruta **/**), probablemente la pantalla de inicio de sesión o el home de la aplicación.

**8.5. catch (error) { console.error('Error al actualizar last\_login:', error); }**

* **Lo que hace**:
  + Si ocurre algún error durante el proceso de actualización del **last\_login** en el servidor, se captura el error y se muestra en la consola.

**9. return (**

* **Lo que hace**:
  + Retorna el contenido del componente **Footer**, que se renderizará en la interfaz del usuario.

**10. <footer className="footer">**

* **Lo que hace**:
  + Renderiza un elemento HTML **<footer>** que contiene el pie de página de la aplicación.
  + **className="footer"**: Aplica la clase de estilo **footer** para el diseño del pie de página (probablemente usando CSS).

**11. <div className="container">**

* **Lo que hace**:
  + Crea un contenedor para los enlaces y el botón de logout.

**12. <Link to="/about">About Us</Link>**

* **Lo que hace**:
  + Crea un enlace de navegación a la página **/about**, usando el componente **Link** de **react-router-dom**.

**13. <Link to="/social">Social</Link>**

* **Lo que hace**:
  + Crea un enlace de navegación a la página **/social**.

**14. <Link to="/contact">Contact Us</Link>**

* **Lo que hace**:
  + Crea un enlace de navegación a la página **/contact**.

**15. {userRole !== 'guest' && ( ... )}**

* **Lo que hace**:
  + Comprueba si el rol del usuario no es **guest**.
  + Si el usuario no es invitado, se mostrará un botón para cerrar sesión (logout).

**16. <button onClick={handleLogout} className="btn btn-link">Log Out</button>**

* **Lo que hace**:
  + Renderiza un botón de **Log Out** que llama a la función **handleLogout** cuando se hace clic en él.
  + La clase **btn btn-link** es un estilo de **Bootstrap** que le da apariencia de enlace al botón.

**17. export default Footer;**

* **Lo que hace**:
  + Exporta el componente **Footer** para que pueda ser utilizado en otras partes de la aplicación.

**Resumen General:**

Este componente **Footer** incluye enlaces a varias secciones de la aplicación y un botón de logout. Cuando se hace clic en **Log Out**, la aplicación actualiza el campo **last\_login** del usuario en la base de datos, limpia el **localStorage** y redirige al usuario a la página principal de la aplicación. Solo los usuarios que no tienen el rol de "guest" ven el botón de logout.

Lastlogin.php

Texto

Descripción generada automáticamente

**$data = json\_decode(file\_get\_contents("php://input"), true);**

* **Lo que hace**:
  + **json\_decode()** decodifica los datos que fueron enviados desde el frontend en formato **JSON** y los convierte en un array asociativo de PHP.
  + **file\_get\_contents("php://input")** obtiene los datos en bruto de la solicitud **POST**.
  + El segundo parámetro **true** en **json\_decode** asegura que el resultado sea un array asociativo (clave-valor) en lugar de un objeto.

**3. $email = $data['email'];**

* **Lo que hace**:
  + Extrae el valor del **email** de los datos recibidos desde el **frontend** (probablemente del **localStorage**).
  + **$email** contiene la dirección de correo electrónico del usuario que está cerrando sesión, que será utilizada para identificar qué usuario necesita que se actualice su campo **last\_login**.

**4. // Actualizar el last\_login con la fecha y hora actual**

* **Lo que hace**:
  + Este comentario explica lo que hará la siguiente consulta SQL: actualizar el campo **last\_login** con la fecha y hora actual.

**5. $sql = "UPDATE User SET last\_login = NOW() WHERE email = ?";**

* **Lo que hace**:
  + Define la consulta SQL para actualizar el campo **last\_login** en la tabla **User**.
  + **NOW()** es una función de MySQL que devuelve la fecha y la hora actuales del servidor.
  + **WHERE email = ?** indica que se actualizará el campo **last\_login** solo para el usuario cuyo **email** coincida con el valor proporcionado.

**6. $stmt = $conn->prepare($sql);**

* **Lo que hace**:
  + Prepara la consulta SQL definida en **$sql** para su ejecución.
  + **$conn** es la conexión a la base de datos, que debería haber sido definida antes (no se muestra en este fragmento).

**7. $stmt->bind\_param("s", $email);**

* **Lo que hace**:
  + Vincula el parámetro **email** a la consulta preparada.
  + **"s"** indica que el tipo de parámetro es una cadena de texto (string).
  + **$email** es el valor que reemplaza al signo de interrogación **?** en la consulta SQL.

**8. $stmt->execute();**

* **Lo que hace**:
  + Ejecuta la consulta SQL preparada.
  + En este punto, la consulta **UPDATE** se envía a la base de datos para actualizar el campo **last\_login** del usuario correspondiente.

**9. if ($stmt->affected\_rows > 0) { ... }**

* **Lo que hace**:
  + Verifica si la consulta ha afectado a alguna fila de la tabla, es decir, si la actualización fue exitosa.
  + **$stmt->affected\_rows** devuelve el número de filas afectadas por la consulta.

**9.1. echo json\_encode(['message' => 'Last login updated successfully']);**

* **Lo que hace**:
  + Si la consulta actualizó correctamente el campo **last\_login**, se envía una respuesta en formato **JSON** al **frontend** con un mensaje de éxito.

**9.2. else { echo json\_encode(['error' => 'Failed to update last login']); }**

* **Lo que hace**:
  + Si no se actualizó ninguna fila (por ejemplo, si el email no coincide con ningún usuario), se envía una respuesta en **JSON** indicando que no se pudo actualizar el campo **last\_login**.

**10. $stmt->close();**

* **Lo que hace**:
  + Cierra la declaración preparada, liberando los recursos utilizados por la misma.

**11. $conn->close();**

* **Lo que hace**:
  + Cierra la conexión a la base de datos, finalizando el uso de la base de datos para esta solicitud.

**Resumen General:**

Este script actualiza el campo **last\_login** del usuario identificado por su **email**, estableciéndolo a la fecha y hora actuales cuando el usuario cierra sesión. Si la actualización es exitosa, el script responde con un mensaje de éxito en formato **JSON**. Si no, responde con un mensaje de error.

APP.js  
Captura de pantalla de computadora

Descripción generada automáticamente

**const userRole = localStorage.getItem('userRole');**

* **Qué hace**:
  + **localStorage.getItem('userRole')** obtiene el valor de **userRole** (el rol del usuario) almacenado en el **localStorage** del navegador.
  + **userRole** es una variable que contiene el rol actual del usuario logueado (por ejemplo, puede ser 1 para un usuario normal, 2 para un empleado, o 3 para un administrador).

**2. const ProtectedRoute = ({ children, allowedRoles }) => { ... };**

* **Qué hace**:
  + Esta es una **función de componente** que se utiliza para proteger rutas según el rol del usuario.
  + Recibe dos parámetros:
    1. **children**: Representa los componentes que se mostrarán dentro de esa ruta protegida (por ejemplo, el componente **Admin**, **Emp** o **User**).
    2. **allowedRoles**: Un array de roles permitidos para acceder a la ruta (por ejemplo, [1, 2, 3] para que los usuarios con esos roles puedan acceder).

**2.1. allowedRoles.includes(parseInt(userRole))**

* **Qué hace**:
  + **allowedRoles.includes()** verifica si el **rol del usuario** (**userRole**) está dentro de los roles permitidos.
  + **parseInt(userRole)** convierte el valor de **userRole** en un número entero, ya que el rol puede estar almacenado como cadena en el **localStorage**.

**2.2. ? children : <Navigate to="/" />**

* **Qué hace**:
  + Si el rol del usuario está dentro de los **roles permitidos** (**allowedRoles**), entonces se renderizan los **children** (el contenido protegido).
  + Si el rol no está permitido, se redirige al usuario a la página principal **/** utilizando el componente **Navigate**.

**3. return (<BrowserRouter>... </BrowserRouter>);**

* **Qué hace**:
  + El componente **BrowserRouter** envuelve todas las rutas de la aplicación y permite manejar la navegación.

**4. Routes y Route**

* **Qué hacen**:
  + **<Routes>** define un conjunto de rutas que pueden ser navegadas en la aplicación.
  + **<Route path='...' element={...}** define una ruta individual y lo que se debe renderizar cuando se accede a esa ruta.

**4.1. Route path='/' element={ <Login/>}**

* **Qué hace**:
  + Define que cuando el usuario navegue a la ruta **/**, se renderizará el componente **Login**.

**4.2. Route path='/signup' element={ <Signup/>}**

* **Qué hace**:
  + Define que cuando el usuario navegue a **/signup**, se renderizará el componente **Signup**.

**4.3. Route path='/forgotpassword' element={ <Forgotpassword/>}**

* **Qué hace**:
  + Define que cuando el usuario navegue a **/forgotpassword**, se renderizará el componente **Forgotpassword**.

**5. Rutas protegidas con ProtectedRoute**

* **Qué hacen**:
  + Las siguientes rutas están protegidas y solo pueden ser accedidas si el usuario tiene un rol permitido.

**5.1. Route path="/admin" element={<ProtectedRoute allowedRoles={[1, 2, 3]}><Admin /></ProtectedRoute>}**

* **Qué hace**:
  + La ruta **/admin** solo puede ser accedida por usuarios con los roles **1, 2 o 3**.
  + Si el usuario tiene alguno de esos roles, se renderizará el componente **Admin**. De lo contrario, será redirigido a la página principal **/**.

**5.2. Route path="/emp" element={<ProtectedRoute allowedRoles={[2]}><Emp /></ProtectedRoute>}**

* **Qué hace**:
  + La ruta **/emp** solo puede ser accedida por usuarios con el rol **2** (por ejemplo, empleados).
  + Si el usuario tiene este rol, se renderizará el componente **Emp**. De lo contrario, será redirigido a la página principal **/**.

**5.3. Route path="/user" element={<ProtectedRoute allowedRoles={[1]}><User /></ProtectedRoute>}**

* **Qué hace**:
  + La ruta **/user** solo puede ser accedida por usuarios con el rol **1**.
  + Si el usuario tiene este rol, se renderizará el componente **User**. De lo contrario, será redirigido a la página principal **/**.

**Resumen:**

Este código implementa una protección de rutas basada en roles de usuario. Dependiendo del rol almacenado en el **localStorage** del navegador, el usuario puede acceder a rutas específicas (como **admin**, **emp**, **user**). Si no tiene el rol adecuado, será redirigido a la página principal.

Header

Texto

Descripción generada automáticamente

**1. const userRole = parseInt(localStorage.getItem('userRole'));**

* **Qué hace**:
  + Obtiene el valor de **userRole** (rol del usuario) almacenado en el **localStorage**.
  + **parseInt** convierte el rol en un número entero, ya que en **localStorage** los valores se almacenan como cadenas.
  + Este valor se usa más adelante para determinar qué opciones se mostrarán según el rol del usuario.

**2. <header className="header">**

* **Qué hace**:
  + Aquí se define la sección del encabezado de la página.
  + Contiene el logo, la barra de búsqueda, y enlaces a otras funcionalidades.

**2.1. <img src="..." alt="Logo" className="logo" />**

* **Qué hace**:
  + Muestra una imagen que actúa como el logo de la aplicación (en este caso, una imagen de GitHub).

**2.2. <input type="text" className="search-bar" placeholder="Search" />**

* **Qué hace**:
  + Es un campo de entrada (input) que representa una barra de búsqueda. Los usuarios pueden escribir aquí para buscar productos en la página.

**2.3. <div className="account-and-cart">**

* **Qué hace**:
  + Agrupa dos secciones:
    1. **Enlace a "My Account" y "My Order"**: Solo se muestran si el usuario tiene un rol diferente de **guest** (invitado). Esto se verifica usando la condición **userRole !== 'guest'**.
    2. **Icono del carrito de compras**: Siempre se muestra, independientemente del rol.

**3. My Account y My Order**

* **Qué hacen**:
  + Estos enlaces permiten al usuario acceder a sus datos personales y a sus órdenes. Se muestran solo si el usuario tiene un rol distinto al de **guest** (usuario no registrado).

Copy code

{userRole !== 'guest' && <Link to="/Account" className="account-link">My Account</Link>}

{userRole !== 'guest' && <Link to="/order" className="order-link">My Order</Link>}

**4. <img src="..." alt="Shopping Cart" className="cart-icon" />**

* **Qué hace**:
  + Muestra un ícono de carrito de compras, el cual es un acceso directo al carrito del usuario.

**5. <nav className="nav-bar">**

* **Qué hace**:
  + Define la barra de navegación que contiene enlaces a diferentes secciones de la aplicación.

**5.1. Home y All Products**

* **Qué hacen**:
  + Son enlaces visibles para todos los usuarios, independientemente del rol. Permiten navegar a la página de inicio y a la página que lista todos los productos.

**5.2. Opciones para empleados y administradores**

jsx

Copy code

{userRole >= 2 && (

<>

<Link to="/manage-orders" className="nav-link">Manage Orders</Link>

<Link to="/inventory" className="nav-link">Inventory</Link>

</>

)}

* **Qué hacen**:
  + Estas opciones se muestran si el rol del usuario es **2** o mayor (empleado o administrador).
  + Incluyen la gestión de órdenes y la gestión de inventario.

 **Header**: Muestra el logo, una barra de búsqueda y un ícono de carrito de compras.

 **Condicionales basadas en userRole**:

* Si el usuario tiene un rol distinto de **guest**, se muestran los enlaces a **My Account** y **My Order**.
* Si el usuario es **empleado** o **administrador**, se muestran enlaces adicionales para la gestión de órdenes e inventario.
* Si el usuario es **administrador**, se muestran opciones de gestión avanzadas como la administración de usuarios, categorías y productos.

Home.js

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

Texto

Descripción generada automáticamente

 **Objetivo**: Esta función divide un array en subarrays más pequeños de tamaño fijo (size).

 **Parámetros**:

* array: El array que quieres dividir.
* size: El número de elementos que cada subarray tendrá.

 **Funcionamiento**:

* La función recorre el array original, tomando "trozos" de size elementos y los añade a result.
* Usa slice(i, i + size) para obtener una porción del array desde el índice i hasta i + size (no incluye i + size).
* La función devuelve el result, que es una lista de subarrays.

 **Uso**: En este código, esta función se usa para dividir la lista de productos más vendidos en grupos de 4 productos.

Texto

Descripción generada automáticamente

 **Header**: Muestra el componente de encabezado.

 **main**: La sección principal del contenido de la página. Aquí se incluyen distintas partes como el video de presentación y los productos más vendidos.

 **VideoComponent**: Este componente muestra un video en la sección "About Us". Recibe la URL del video a través de la prop videoUrl.

Texto

Descripción generada automáticamente

 **Carrusel (Carousel)**: Un componente de **react-bootstrap** que permite desplazarse entre grupos de productos.

 **chunkArray(topProducts, 4)**: Divide el array de productos más vendidos (topProducts) en grupos de 4.

 **productGroup.map**: Recorre cada grupo de productos y genera una tarjeta (Card) para cada producto.

 **Card**: Cada tarjeta muestra un producto con:

* Una imagen del producto (Card.Img).
* El nombre del producto (Card.Title).
* El precio del producto (Card.Text).
* Un botón que dice "Ver Artículo" debajo del producto.

 **Diseño**: Las tarjetas están dentro de una estructura de carrusel donde los productos se muestran en grupos de 4. La clase d-flex justify-content-around alinea las tarjetas horizontalmente.

Texto

Descripción generada automáticamente

 **categories.map**: Recorre el array de categorías y genera una tarjeta (Card) para cada una.

 **Card.Title**: Muestra el nombre de cada categoría dentro de la tarjeta.

 **Diseño**: Las categorías se presentan en una estructura tipo grid, con tarjetas para cada una. La clase card-grid probablemente se usa para distribuir las tarjetas uniformemente en la pantalla.

Category

Texto

Descripción generada automáticamente

**Form, Row, Col, Card, Button**: Componentes de **react-bootstrap**, utilizados para generar formularios, tarjetas, botones, y para organizar el contenido en filas y columnas.

Texto

Descripción generada automáticamente

**<Col md={4}>**

Esta etiqueta está usando un **componente de Bootstrap** llamado Col, que es una columna. Dentro de la rejilla de Bootstrap, las columnas están divididas en 12 partes. El atributo md={4} significa que esta columna ocupará 4 de las 12 columnas disponibles en pantallas medianas o superiores. Por lo tanto, si hay tres columnas de este tipo en una fila (3 x 4 = 12), llenarán todo el ancho de la pantalla.

**2. <Form.Group controlId="priceFilter">**

El componente **Form.Group** es usado para agrupar elementos de formulario en Bootstrap. El atributo controlId="priceFilter" es un identificador único que se utiliza para vincular el Form.Group con su etiqueta y sus elementos de formulario.

**3. <Form.Label>Price Range</Form.Label>**

Aquí se define una **etiqueta** (label) para el selector de rangos de precios. Este texto ("Price Range") será visible encima del desplegable, indicando que los usuarios pueden seleccionar un rango de precios.

**4. <Form.Control as="select">**

El componente **Form.Control** con el atributo as="select" renderiza un **desplegable** (dropdown). El desplegable permite a los usuarios elegir entre varias opciones. En este caso, las opciones representan rangos de precios.

* **El desplegable** es el elemento que contendrá las diferentes opciones que los usuarios pueden seleccionar, y las opciones se renderizan dentro de este.

**5. <option value="">All Prices</option>**

Esta es la primera opción fija del desplegable. El valor value="" indica que no se ha seleccionado ningún rango de precio en particular, lo que generalmente se interpreta como "mostrar todos los productos" sin importar el precio. El texto "All Prices" es lo que el usuario verá en la lista desplegable.

**6. Mapeo con .map()**

javascript

Copy code

{priceRanges.map((range, index) => (

<option key={index} value={range}>{range}</option>

))}

Esta parte es la que realiza el **mapeo dinámico** de los datos del array priceRanges y genera una lista de opciones en el desplegable.

* **priceRanges**: Es un array que contiene los diferentes rangos de precios, como ['Under $100', '$100 - $200', '$200 - $300', 'Above $300'].
* **priceRanges.map()**: El método .map() itera sobre cada elemento del array priceRanges. Por cada elemento del array, se ejecuta una función de devolución (callback) que genera un elemento <option>.
  + **range**: Este es cada valor individual del array priceRanges, como 'Under $100' o '$100 - $200'.
  + **index**: Es el índice del elemento actual en el array. Se utiliza para proporcionar una propiedad key única a cada <option>. Esto es importante en React para optimizar la representación y el seguimiento de los elementos en la interfaz.

**7. Renderizado de cada <option>**

Dentro del mapeo, por cada elemento del array, se genera lo siguiente:

javascript

Copy code

<option key={index} value={range}>{range}</option>

* **<option>**: Cada iteración genera una etiqueta <option>, que es un elemento individual en el desplegable.
* **key={index}**: React necesita que cada elemento de una lista tenga una key única. En este caso, estamos usando el índice del array como key (que es seguro aquí porque los elementos no cambian de orden).
* **value={range}**: El valor del atributo value será el valor del rango de precios, como '$100 - $200'. Cuando un usuario selecciona esta opción, el valor que se selecciona es el rango de precios.
* **{range}**: Este es el texto que el usuario verá en el desplegable. Por ejemplo, mostrará 'Under $100', '$100 - $200', etc.

**8. Ejemplo de lo que genera el mapeo**

Después del mapeo, las opciones renderizadas dentro del desplegable se verán así:

html

Copy code

<option value="Under $100">Under $100</option>

<option value="$100 - $200">$100 - $200</option>

<option value="$200 - $300">$200 - $300</option>

<option value="Above $300">Above $300</option>

Esto permite que los usuarios seleccionen uno de los rangos de precios cuando quieran filtrar productos según su precio.

**Resumen**

* El **mapeo (.map())** toma los elementos del array priceRanges y genera dinámicamente las opciones dentro del desplegable de rango de precios.
* **Cada <option>** tiene un valor único basado en el rango de precios, y los usuarios pueden seleccionar cualquiera de esos rangos.
* React utiliza el atributo key para identificar de manera única cada opción renderizada, lo que mejora el rendimiento y la gestión de las listas.