Manual Técnico

Backend:

Routes

AuthRoute

1. **Importación de Módulos**:

javascript

```
const express = require('express');

const router = express.Router();
```

   - Se importa el módulo express y se crea un nuevo router utilizando express.Router(). Este router se utilizará para definir las rutas de la API.

2. **Definición de una Constante**:

javascript

```
const carnet = '202100171';
```

   - Se define una constante carnet, aunque en este fragmento de código no se utiliza. Podría ser un identificador o número de carnet para propósitos futuros.

3. **Manejo de la Ruta /login**:

javascript

```
router.post('/login', (req, res) => {
  const { username, password } = req.body;
  if (username === `1` && password === `1`) {
    res.status(200).json({ message: 'Login successful' });
  } else {
    res.status(401).json({ message: 'Invalid credentials' });
  }
});
```

   - Se define un endpoint que responde a las solicitudes POST en la ruta /login.
   - Se extraen username y password del cuerpo de la solicitud (req.body).
   - Se verifica si el username y password son ambos iguales a 1. Si es así, se responde con un estado 200 (OK) y un mensaje de éxito en formato JSON.
   - Si las credenciales son incorrectas, se responde con un estado 401 (Unauthorized) y un mensaje de error.

4. **Exportación del Router**:

javascript

module.exports = router;

- Finalmente, se exporta el router para que pueda ser utilizado en otras partes de la aplicación, como en el archivo principal del servidor.

ClientRoutes.js

Estructura del Código

Importaciones

javascript

```
const express = require('express');

const router = express.Router();

const dataStore = require('../data/dataStore');
```

- **express**: Importa el framework Express para crear el servidor y manejar las rutas.
- **router**: Crea un enrutador para definir las rutas de la API.
- **dataStore**: Importa un módulo que contiene la estructura de datos donde se almacenan los clientes.

Rutas Definidas

1. Crear Cliente

javascript

```
router.post('/', (req, res) => {
  const { id_cliente, nombre_cliente, apellido_cliente, nit_cliente, edad } = req.body;


  if (dataStore.clients.some(c => c.id_cliente === id_cliente)) {
    return res.status(400).json({ message: 'Client already exists' });
  }


  const newClient = { ...req.body, nit: nit_cliente || 'C/F' };
  dataStore.clients.push(newClient);
  res.status(201).json({ message: 'Client added successfully', clients: dataStore.clients });
});
```

- **Método**: POST
- **Ruta**: /
- **Descripción**: Crea un nuevo cliente. Verifica si el cliente ya existe utilizando su id_cliente. Si no existe, se agrega a la lista de clientes y se devuelve un mensaje de éxito junto con la lista actualizada de clientes.
- **Respuestas**:

- 201: Cliente agregado exitosamente.
- 400: El cliente ya existe.

2. Listar Clientes

javascript

```
router.get('/', (req, res) => {
  res.status(200).json(dataStore.clients);
});
```

- **Método**: GET
- **Ruta**: /
- **Descripción**: Devuelve la lista de todos los clientes almacenados en dataStore.
- **Respuesta**: 200: Lista de clientes.

3. Eliminar Cliente

javascript

```
router.delete('/:id', (req, res) => {
  const { id } = req.params;
  dataStore.clients = dataStore.clients.filter(c => c.id_cliente !== id);
  res.status(200).json({ message: 'Client deleted successfully', clients: dataStore.clients });
});
```

- **Método**: DELETE
- **Ruta**: /:id
- **Descripción**: Elimina un cliente basado en su id_cliente. Filtra la lista de clientes para excluir al cliente que se desea eliminar y devuelve un mensaje de éxito junto con la lista actualizada.
- **Respuesta**: 200: Cliente eliminado exitosamente.

Exportación del Router

javascript

```
module.exports = router;
```

- **Descripción**: Exporta el enrutador para que pueda ser utilizado en otras partes de la aplicación, como en el archivo principal del servidor.

DataStore.js

Funcionalidades

1. Almacenamiento de Productos

- **Descripción**: Permite almacenar productos en el array products.
- **Uso**: Se puede agregar un nuevo producto al array mediante operaciones de inserción en el backend.

2. Almacenamiento de Clientes

- **Descripción**: Permite almacenar clientes en el array clients.
- **Uso**: Se puede agregar un nuevo cliente al array mediante operaciones de inserción en el backend.

Integración

Este módulo se exporta para ser utilizado en otros archivos de la aplicación, permitiendo que diferentes partes del sistema accedan y modifiquen los datos de productos y clientes de manera centralizada.

Frontend React

LoginPage.js

1. **Importaciones**:
    - Se importan React, useState, useNavigate de react-router-dom y axios para manejar el estado y las solicitudes HTTP.
2. **Estado del Componente**:
    - username: Almacena el nombre de usuario ingresado.
    - password: Almacena la contraseña ingresada.
    - error: Almacena mensajes de error en caso de que las credenciales sean incorrectas.
3. **Función handleLogin**:
    - Se ejecuta al enviar el formulario. Realiza una solicitud POST a la API de autenticación.
    - Si la respuesta es exitosa (código 200), redirige al usuario al panel de control.
    - Si hay un error, se actualiza el estado de error para mostrar un mensaje.
4. **Renderizado**:
    - Se muestra un formulario con campos para el nombre de usuario y la contraseña.
    - Si hay un error, se muestra un mensaje en rojo.

Dashboard.js

1. **Importaciones**:

- Se importan varios componentes que se utilizarán en el dashboard, incluyendo tablas para productos y clientes, así como formularios para crear nuevos clientes y productos, y gráficos para visualizar datos.

2. **Estructura del Componente**:
   - El componente DashboardPage utiliza un contenedor principal que aplica un estilo de padding.
   - Se incluye un encabezado que indica que se trata del "Cobra Kai Dashboard".
3. **Diseño de la Interfaz**:
   - **Crear Cliente y Crear Producto**: Se presentan botones para crear nuevos clientes y productos en la parte superior del dashboard.
   - **Tablas de Productos y Clientes**: Se muestran tablas que permiten visualizar y gestionar los productos y clientes existentes.
   - **Gráficos**: Se incluyen gráficos de barras y de pastel para representar visualmente los datos de productos y clientes.
4. **Estilos**:
   - Se utiliza flexbox para organizar los componentes en filas, asegurando que se distribuyan de manera uniforme y responsiva.

ProducTable.js

1. **Importaciones**:
   - Se importan React, useState, y useEffect de la biblioteca de React, así como axios para realizar solicitudes HTTP.
2. **Estado del Componente**:
   - Se define un estado products que almacenará la lista de productos obtenidos desde el servidor.
3. **Efecto para Obtener Productos**:
   - Se utiliza useEffect para realizar una solicitud GET a la API cuando el componente se monta. Esto permite cargar los productos desde http://localhost:3001/products y actualizar el estado con los datos recibidos.
4. **Manejo de Eliminación**:
   - La función handleDelete se encarga de eliminar un producto específico. Realiza una solicitud DELETE a la API y actualiza el estado para eliminar el producto de la lista local.
5. **Renderizado de la Tabla**:
   - Se renderiza una tabla que muestra los productos. Cada fila incluye el ID, nombre, precio, stock y un botón para eliminar el producto.

CreateProduct.js

1. Estado del Componente

- **useState**: Se utilizan dos estados:
    - productJson: Almacena el texto JSON ingresado por el usuario que representa los datos del producto.
    - message: Almacena mensajes de éxito o error que se mostrarán al usuario después de intentar crear un producto.

2. Función handleCreateProduct

- **Propósito**: Esta función se encarga de manejar la creación de un nuevo producto.
- **Proceso**:
    - **Parseo del JSON**: Intenta convertir el texto ingresado en productJson a un objeto JavaScript utilizando JSON.parse().
    - **Solicitud POST**: Realiza una solicitud HTTP POST a la API en http://localhost:3001/products con los datos del producto.
    - **Manejo de Respuesta**: Si la respuesta es exitosa (código de estado 201), se actualiza el mensaje de éxito, se limpia el campo de entrada y se llama a fetchProducts() para actualizar la lista de productos en la tabla.
    - **Manejo de Errores**: Si ocurre un error durante el proceso, se captura y se muestra un mensaje de error.

3. Renderizado del Componente

- **Estructura HTML**: El componente renderiza un contenedor con un título, un área de texto para ingresar el JSON del producto, un botón para crear el producto y un párrafo para mostrar mensajes.
- **textarea**: Permite al usuario ingresar el JSON del producto. Su valor está vinculado al estado productJson, y se actualiza mediante onChange.
- **Botón de Crear Producto**: Al hacer clic en este botón, se ejecuta la función handleCreateProduct.

PieChart.js

Importaciones

- **React y Hooks**: Se importan React, useEffect, y useState para manejar el estado y los efectos secundarios en el componente.
- **Recharts**: Se importan componentes de la biblioteca recharts para crear gráficos, específicamente PieChart, Pie, Cell, Tooltip, y ResponsiveContainer.
- **Axios**: Se importa axios para realizar solicitudes HTTP.

Estado del Componente

- **data**: Se define un estado llamado data utilizando useState, que se inicializa como un array vacío. Este estado almacenará los datos que se obtendrán de la API.

Efecto Secundario

- **useEffect**: Se utiliza useEffect para ejecutar una función cuando el componente se monta. Esta función se encarga de:
    - **fetchClientsStats**: Realiza una solicitud GET a la API para obtener estadísticas de clientes. Si la solicitud es exitosa, se estructura la respuesta en un formato adecuado (un array de objetos con name y value) y se actualiza el estado data con estos valores.
    - **Manejo de Errores**: Si ocurre un error durante la solicitud, se captura y se imprime en la consola.

Colores para el Gráfico

- **COLORS**: Se define un array de colores que se utilizarán para diferenciar las secciones del gráfico circular.

Renderizado del Componente

- **Estructura del Gráfico**: El componente retorna un div que contiene:
    - Un encabezado que indica que se trata de un gráfico de clientes por edad.
    - Un ResponsiveContainer que asegura que el gráfico se ajuste al tamaño del contenedor.
    - Un PieChart que incluye:
        - Un Pie que utiliza los datos del estado data, especificando cómo se deben mostrar los datos (con dataKey y nameKey).
        - Un mapeo de los datos para crear celdas (Cell) con colores específicos.
        - Un Tooltip que muestra información adicional al pasar el cursor sobre las secciones del gráfico.